Model Driven Software Modernisation

PhD Thesis

Feng Chen

Software Technology Research Laboratory

De Montfort University

2007

To my wife, Miaomiao Liu,
my son, Qiyuan Chen and
my parents.

Declaration

I declare that the work described in this thesis was originally carried out by me during the period of registration for the degree of Doctor of Philosophy at De Montfort University, U.K., from April 2003 to March 2007. It is submitted for the degree of Doctor of philosophy at De Montfort University. Apart from the degree that this thesis is currently applying for, no other academic degree or award was applied for by me based on this work.

Acknowledgements

For many years I had been dreaming about receiving a PhD. I would like to thank many people who helped me in achieving this dream in different ways when I undertook the work of this thesis.

I wish to express my most profound thanks to my supervisors, Prof. Hongji Yang and Prof. He Guo, for their invaluable advice, experienced guidance and encouragement during my four-year study. They provided me with many useful comments and suggestions for preparation of this thesis.

My thanks must go to Prof. Hussein Zedan, the director of the laboratory, for providing many helpful suggestions during our many discussions. My research career benefits tremendously from the research methodologies that Prof. Hussein Zedan introduced to me.

I wish to thank Prof. Malcolm Munro and Prof. Hussein Zedan for examining the thesis carefully and the invaluable comments they gave to me.

I would also like to thank the Research Office at De Montfort University for their outstanding management.

I wish to express thanks to my wife, Miaomiao Liu, my parents and parents in law for all their loves, encouragements, patience and supports over the years. This thesis is dedicated to them.

Finally, I would like to thank all of the colleagues in Software Technology Research Laboratory at De Montfort University and the colleagues in Dalian University of Technology, for their valuable suggestions and discussions, for their encouragement and support, for building great software packages, including: Shaoyun Li, Matthias Ladkau, Stefan Natelberg, Peer Bartels, Yuxin Wang, and many others. I am indebted to all of them.

Abstract

Constant innovation of information technology and ever-changing market requirements relegate more and more existing software to legacy status. Generating software through reusing legacy systems has been a primary solution and software re-engineering has the potential to improve software productivity and quality across the entire software life cycle. The classical re-engineering technology starts at the level of program source code which is the most or only reliable information on a legacy system. The program specification derived from legacy source code will then facilitate the migration of legacy systems in the subsequent forward engineering steps. A recent research trend in re-engineering area carries this idea further and moves into model driven perspective that the specification is presented with models.

The thesis focuses on engaging model technology to modernise legacy systems. A unified approach, REMOST (Re-Engineering through MOdel conStruction and Transformation), is proposed in the context of Model Driven Architecture (MDA). The theoretical foundation is the construction of a WSL-based Modelling Language, known as WML, which is an extension of WSL (Wide Spectrum Language). WML is defined to provide a spectrum of models for the system re-engineering, including Common Modelling Language (CML), Architecture Description Language (ADL) and Domain Specific Modelling Language (DSML). *Meta*WML is designed for model transformation, providing query facilities, action primitives and metrics functions. A set of transformation rules are defined in *Meta*WML to conduct system abstraction and refactoring. Model transformation for unifying WML and UML is also provided, which can bridge the legacy systems to MDA. The architecture and working flow of the REMOST approach are proposed and a prototype tool environment is developed for testing the approach. A number of case studies are used for experiments with the approach and the prototype tool, which show that the proposed approach is feasible and promising in its domain. Conclusion is drawn based on analysis and further research directions are also discussed.

Table of Contents

List of Figures

List of Tables

List of Lists

List of Acronyms

AADL	*Agent-oriented Architecture Description Language*
ADL	*Architecture Description Language*
ADM	*Architecture Driven Modernisation*
AOP	*Aspect Oriented Programming*
API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
BNF	*Backus-Naur Form*
CIM	*Computation Independent Model*
CML	*Common Modelling Language*
DSML	*Domain Specific Modelling Language*
EBNF	*Extended Backus-Naur Form*
EJBs	*Enterprise JavaBeans*
EMF	*Eclipse Modelling Framework*
ER	*Entity-Relationship*
FCFS	*First-Come-First-Served*
FDL	*Feature Description Language*
FIP	*FermaT Integration Platform*
FME	*FermaT Moderniser's Environment*
FODA	*Feature Oriented Domain Analysis*
FORM	*Feature-Oriented Reuse Method*
TAGDUR	*Transformation and Automatic Generation of Documentation in UML through Re-engineering*
GME	*Generic Modelling Environment*
GUI	*Graphic User Interface*
ICE	*In-Circuit Emulator*
IDL	*Interface Description Language*
ITL	*Interval Temporal Logic*

LOC	*Line Of Code*
MA	*Maintainer's Assistant*
MAS	*Multi-Agent System*
MDA	*Model Driven Architecture*
MDD	*Model Driven Development*
MDE	*Model Driven Engineering*
MDRE	*Model Driven RE-engineering*
MIC	*Model Integrated Computing*
MIPS	*Model Integrated Program Synthesis*
MIS	*Management Information System*
MMLs	*Modelling Maturity Levels*
MOF	*Meta-Object Facility*
MVC	*Model-View-Controller*
NFR	*Non-Functional Requirement*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OORSL	*Object-Oriented Requirements Specification Language*
PIM	*Platform Independent Model*
POSIX	*Portable Operating System Interface*
PSM	*Platform Specific Model*
QVT	*Query/View/Transformation*
REMOST	*Re-Engineering through Model conStruction and Transformation*
RFP	*Request For Proposal*
RMI	*Remote Method Invocation*
RTOS	*Real-Time Operating System*
SDL	*Specification and Description Language*
SERG	*Software Evolution Research Group*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
SQL	*Structured Query Language*
STRL	*Software Technology Research Laboratory*

TAM	*Temporal Agent Model*
TS	*Technological Space*
UDDI	*Universal Description Discovery and Integration*
UML	*Unified Modelling Language*
VOS	*Virtual Operating System*
VRTOS	*Virtual Real-Time Operating System*
WSL	*Wide Spectrum Language*
WML	*WSL-based Modelling Language*
WSDL	*Web Service Description Language*
WSW	*Web Services Wrapper*
XMI	*XML Meta-data Interchange*
XML	*eXtensible Markup Language*
URI	*Uniform Resource Identifier*

Chapter 1
Introduction

Objectives

- To motivate the need for model driven software modernisation.
- To explain the research characteristics and select the research method.
- To identify the research questions and illustrate the research hypothesis.
- To highlight original contribution and define the success criteria.

1.1 Motivation

Constant innovation of information technology and ever-changing market requirements relegate more and more existing software to legacy status. From the moment a software product is released, the race against time and aging begins. Organisations are in fear of their legacy systems. They are afraid of keeping them, since maintaining them is a significant drain on the organisation's resources. They are also afraid of replacing them. A major reason is that those legacy systems are enormously valuable assets. Having stood the test of time and evolved, they provide the most accurate statement of current business practices. Legacy systems, then, are not an issue that can be simply thrown away.

Software modernisation attempts to evolve a legacy system, or elements of the system, when conventional evolutionary practices, such as maintenance and enhancement, can no longer achieve the desired system properties [142]. Software

modernisation falls between the two extremes of system replacement and continued maintenance. Software re-engineering is a form of modernisation that improves capabilities and maintainability of a legacy system by introducing modern technologies and practices [142, 171]. The purpose of software re-engineering is both to position existing systems to take advantage of new technologies and to enable new development efforts to take advantage of reusing existing systems. Software re-engineering has the potential to improve software productivity and quality across the entire life cycle.

The classical re-engineering technology starts at the level of program source code that is the most or only reliable information on a legacy system. The program specification derived from legacy source code will then facilitate the migration of legacy systems in the subsequent forward engineering steps. A recent research trend in re-engineering area carries this idea further and moves into model driven perspective which is in the context of Model Driven Architecture (MDA). MDA aims at a unified model based framework for software development. All artifacts, such as requirement specifications, architecture descriptions, design descriptions, and even code, are regarded as models and are represented by modelling languages. A series of models construct a hierarchy of abstraction levels and models of MDA at different abstract levels can be transformed automatically to each other. However, most of the existing approaches to model transformation are limited to the forward engineering only. There is not enough attention for combining traditional software re-engineering techniques and the MDA researches, which is actually of significant importance for successful software evolution. This situation leads to an increasing requirement to carry out model driven software modernisation more efficiently, which triggered the research described in this thesis. The thesis therefore aims to present an approach to re-engineering the legacy system in line with MDA philosophy, including both forward and reverse engineering.

1.2 Research Methods

This section describes the research method applied in this thesis, which links the new knowledge coming from research to the process leading to outcomes. The research field in this thesis belongs to software engineering aiming to be a rigorous discipline and

enable the successful production of software (high quality products at the lowest possible cost). Being a kind of computer science and like all kinds of engineering, the majority of software engineering research is constructive. Constructive research refers to the new contributions being developed. A new contribution can be a new theory, algorithm, model, framework or a method. Since software engineering always involves complex action and interaction of human beings, empirical research is also required to investigate such situation. Hence, the research method applied in this thesis is the combination of empirical and constructive research that is of both high practical utility and academically rigorous. The basic methods used in this thesis are summarised as follows:

- Formal methods: Formal methods can be defined as mathematically based languages, techniques, and tools for specifying and verifying systems. Formal methods can increase the understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might go undetected [25].

- Quantitative and qualitative methods: Both methods are used under the umbrella of the proposed framework, asking Why, What, How, Who, When, Where questions and looking at the problem and solutions from many points of view.

- Modelling: as means of communication, documentation, analysis, simulation, decision making and verification. Modelling can break up a large problem into smaller problems and reduce the complexity.

- Artificial intelligence: when information needed is still missing, artificial intelligence is required.

A framework for the proposed research method is defined in both science and engineering aspects. The research structure and processes were realised as follows:

Step 1: Research Problem, Question and Hypothesis Identification.

The research problem and the motivation for the problem were provided first. Initial understanding of the problem was obtained by literature studies. Previous results related

to the research problem were searched for, studied, and analysed. Resources such as ACM Digital Library, CiteSeer, IEEE Xplore, and SpringerLink were retrieved. Search engines such as Google and Yahoo were used for discovering and crosschecking relevant information. In order to narrow down the research problem, a set of research questions were raised and research hypotheses were formulated to tackle the underlying problem issues.

Step 2: Solution Construction.

Contributions were constructed, which include formalised concepts, a spectrum modelling language and a framework. To demonstrate the applicability of proposed research, a software environment and related tools were developed and applied in both academic and engineering perspectives.

Step 3: Validation and Verification.

The hypotheses were verified by case studies and validated by means of criteria. Case studies were used to validate and demonstrate the feasibility of the proposed approach, which showed that the methodology can work and produce impressive results. It should be mentioned that it is impossible to judge whether the results are entirely due to the methodology used. A challenge in this respect was to carefully select case studies which are representative and which also span the potential application space of the targeted application domain.

Step 4: Conclusions

Based on the experiences from the evaluation, the applicability of the methods was discussed. As research is an iterative process, new relevant and interesting research questions could be found and formed as the future research topics.

1.3 Research Questions

This research is driven by a number of problems inherent in the current state of both academic and industrial research area. These problems inform a rationale for the project.

The overall research question this thesis tries to answer is:

> **How can software models be used to modernise the legacy systems through model construction and transformation?**

In order to be able to answer this question, a set of research questions are defined that address the problem in detail.

RQ1: What is the model driven software modernisation?

RQ2: How can the models be extracted from legacy systems?

- What type of models is required to re-engineer the legacy system?
- How to specify the modelling languages?
- How to link the source code with models?

RQ3: How model transformation can be implemented?

- How to present model transformation and define model transformation rules with model transformation language?
- How to preserve the traceability during the model transformation?
- What is the role of UML in model driven software modernisation?

RQ4: How can tool support be provided for the proposed approach?

1.4 Research Hypothesis

The main hypothesis underlying the present thesis is that software modelling techniques are useful means to manage software modernisation in large complex software systems in a cost efficient manner.

H1: None of current techniques alone is sufficient to prevent the legacy crisis, but integration of various techniques can reduce the overall effort required to maintain the

ever-increasing amount of legacy software code.

H2: The proposed approach adopts a component-based perspective, assuming that legacy systems are componentised through modular or aspectual decomposition, and supporting gradual migration approach.

H3: Since only parts of the re-engineering process can be automated, a cost efficient and highly industrialised re-engineering approach will have to support the followings:

- Provide a well defined re-engineering methodology.
- Maximise automation, but allow for flexibility and combination with manual approaches.
- Provide a decision framework for automation vs. manual re-engineering, based on cost and resulting quality.

H4: Not all parts of a legacy system are of equal value and only small parts of a system are representing real business rules and process logic. In MDA context, the technical infrastructure code needs only be re-engineered into models in coarse granularity.

- This kind of code tends to be very specific to the underlying platform, thus migrating it to a new platform doesn't make a lot of sense. From viewpoint of MDA, this kind of code is easier to be generated from Platform Specific Model (PSM).
- Modern platforms usually provide much technical infrastructure functionality, so there is no need for large amounts of custom "glue" code. Meanwhile, a well structured system has to deal with much fewer exceptional cases, again eliminating the need for custom code that deals with them [67].

1.5 Research Context

The research in this thesis should be seen as a step in a more general effort conducted at the Software Evolution Research Group (SERG) in De Montfort University. The main challenges that SERG have tackled and will continue to tackle are to understand the ever complex software systems and suggest better methods for these systems to evolve and survive to provide the intended services at a lower cost, which builds on two fundamental research achievements under the theme of software evolution.

One fundamental achievement has concentrated on program transformations and their realisation within an industry-strength toolset FermaT [165, 166, 180]. FermaT has applications in forward engineering, reverse engineering and language migration, which is based on the framework of WSL transformations [97, 165, 168, 180]. A unified approach for reverse engineering is described within which the notion of abstraction is classified and precisely defined [97], which attempts to maximise the automation with the assistance of abstraction rules and abstraction pattern assertions [179]. Besides sequential non-time systems, the approach takes time-critical systems with parallelism as its specific application domain. WSL is extended to have real-time and parallel feature: from Interval Temporal Logic (ITL) [109] at specification level to TAM [21] at code level.

Another fundamental achievement is the establishment of a framework known as K-Mediator (here 'K' stands for Knowledge) which is a basis for a sound theory of co-evolution for system design and development [182]. Some of the case studies were based at IBM and some were provided by BT and Abbey National. A three-year research project based on K-Mediator, which was entitled "System Re-engineering using Artificial Intelligence" [95], has been carried out by the investigator at SERG and British Telecommunication. They matched a software program with a pre-defined domain knowledge base in the representation of a simplified semantic network in order to link the source program with its domain-level interpretation [95, 96, 180].

Recently, a number of investigations were relevant to model-based software evolution. In [177], a solution to the acquisition of Entity Relationship Attribute

Diagrams from data-intensive source code was described that could be treated as a pioneer work of model driven re-engineering. In [178], how the obtained ontology could be applied to understanding and eventually better re-engineering the legacy systems was discussed. A dependable enterprise service assembly line for legacy application integration was investigated in [93]. A re-engineering approach which applied an improved agglomerative hierarchical clustering algorithm to restructure legacy code and to facilitate legacy code extraction for Web service construction was proposed in [183]. In [94], a semi-formal parallel requirement specification language OORSL (Object-Oriented Requirements Specification Language) was defined and used to transform the function specification into Java program framework. A prototype tool, TAGDUR, for producing UML diagrams through re-engineering of legacy systems was developed in [107]. A round-trip software re-engineering methodology based on MDA was presented in [19, 131, 132], where traditional program transformation and modern model transformation were combined with WSL.

1.6 Original Contributions

In this thesis, a unified approach, REMOST (Re-Engineering through MOdel conStruction and Transformation), is proposed in the context of MDA, which integrates all technical supports into a systematic method for software modernisation. Concretely, the original contributions of this thesis are as follows:

C1: Wide Spectrum Language (WSL) is extended with a spectrum of modelling languages, called WSL-based Modelling Language (WML), which includes Common Modelling Language (CML), Architecture Description Language (ADL) and Domain Specific Modelling Language (DSML). The work on WSL extensions is done in the wider context of MDA that is a natural continuation of the software transformation framework.

C2: Model transformation language, *Meta*WML, is defined based on *Meta*WSL. A set of query facilities, action primitives and metric functions are established to unify program and model transformation in a seamless way.

C3: A great deal of effort, including the definition of WML and *Meta*WML, is devoted to unifying WML and UML to bridge legacy systems with MDA.

C4: A framework for model construction and transformation based on WML and *Meta*WML is presented. Methodology of model abstraction and refactoring is designed with modern software notions (e.g. design patterns and aspects) that allow for the architecture centred model identification and transformation.

C5: A set of toolsets are developed to demonstrate the effectiveness of the proposed approach by re-engineering demonstrator applications into modern paradigm. Tools are implemented by members in SERG.

1.7 Success Criteria

A whole criterion for the success of REMOST approach is how well they support successful software modernisation. The following criteria are given to judge the success of the research described in this thesis:

- The proposed approach should be able to deal with as many kinds of legacy systems as possible only if the source code of these legacy systems is available.
- The proposed approach should support the modern paradigms like multi-agent systems or Web services.
- The extracted models should be consistent to the original design.
- The extracted models should be unambiguous and easy to understand.
- The extracted models should be reliable to perform forward engineering.
- The proposed approach should be feasible for realisation. For example, it is possible to build a practical tool to demonstrate the approach.
- The proposed approach should be capable for industrial-scaled systems.

1.8 Thesis Outline

The thesis is organised as follows:

Chapter 1 defines the research objectives, explains the research characteristics, selects the research method, identifies the research questions, illustrates the research hypothesis, highlights original contributions and defines the success criteria.

Chapter 2 gives a general overview of research background and defines basic concepts related to model driven software modernisation.

Chapter 3 discusses the related work, which includes software modelling techniques, software architecture, program/model transformation and three projects related to model driven software modernisation.

Chapter 4 introduces a unified approach, REMOST, for model driven software modernisation. The architecture and processes of REMOST are proposed.

Chapter 5 describes the design of WML, which is the extension of WSL with object-oriented modelling techniques, software architecture techniques and domain specific modelling techniques.

Chapter 6 works on a program/model transformation language, called *Meta*WML. A set of query facilities, action primitives and a set of metric functions are established to facilitate program and model transformation.

Chapter 7 describes the methodology of model construction and transformation, including translation from programware to modelware, abstraction from low level models to high level models, model refactoring based on design pattern or aspect, and mappings between WML and UML.

Chapter 8 describes a CASE environment, FermaT Integration Platform (FIP), for supporting the (semi-)automatic software modernisation.

Chapter 9 describes the experiments performed on three case studies, which

demonstrate that the proposed approach works in practice and is indeed scalable. Modern software architecture and paradigms such as Service Oriented Architecture (SOA) and Agent Oriented Programming are covered.

Chapter 10 concludes the thesis. The success criteria are revisited and the future work is discussed.

Appendix A gives the syntax definition of WML.

Appendix B lists commands and utilities of *Meta*WML libraries.

Appendix C lists all the related publications by the author during the PhD study.

Chapter 2

Background and Basic Concepts

Objectives

- To introduce software crisis and legacy crisis.
- To present an overview of software evolution.
- To present an overview of modern software paradigms.
- To define basic concepts related to model driven software modernisation.

2.1 From Software Crisis to Legacy Crisis

No one will doubt today that information systems are business-critical for almost all institutions. However, too much software currently being produced is late, over budget, and does not perform as expected; yet software costs are rising all the time. The fact that the software development industry is in a crisis has been recognised since 1969. Problems associated with the software crisis have been caused by the character of software itself. F. P. Brooks [11] claims the following properties of large software systems:

- Complexity: This is an essential property of all large pieces of software, essential in that it cannot be abstracted away from. This leads to several problems:
  - ✓ Communication difficulties among team members, leading to product flaws, cost overruns, and schedule delays.

- ✓ It is difficult or impossible to enumerate all the states of the system, which makes it impossible to understand the system completely.
- ✓ It is difficult to get an overview of the system, so maintaining conceptual integrity becomes increasingly difficult;
- ✓ It is hard to ensure that all loose ends are accounted for.
- ✓ There is a steep learning curve for new personnel.

- Conformity: Many systems are constrained by the need to conform to complex human institutions and systems (e.g., the tax regulations of a state).
- Change: As it is used any successful system will be subject to change to enhance its capabilities, or even apply it beyond the original domain, as well as to enable it to survive beyond the normal life of the machine it runs on and to be ported to other machines and environments.
- Invisibility: For complex software systems there is no geometric representation, as is available to the designers and builders of complex mechanical or electronic machines or large buildings. There are several distinct but interacting graphs of links between parts of the system to be considered (e.g., control flow, data flow, dependency, and time sequence). One way to simplify these, in an attempt to control the complexity, is to cut links until the graphs become hierarchical structures [125].

As one of the most important areas of computer science, software engineering had its origin as a solution to the "software crisis". According to the IEEE Standards, software engineering is defined as [65]:

Software Engineering *is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.*

Software engineering has three elements: (1) methods, which provide the techniques

for building software including the design of data structures, program architecture, and algorithmic procedure, coding, testing, and maintenance; (2) tools, which provide automated or semi-automated support for methods; and (3) processes, the glue that holds the methods and tools together and enables rational and timely development of computer software. A generic view of software engineering can be obtained by examining the process of software development [180].

Many methods and techniques have been hailed as the solution to the software crisis; however in practice only small gains in productivity have been achieved and few of these methods pay any attention to the problems of maintaining and enhancing the developed software. Estimates show that 65-75% of total software costs are subsumed in maintenance activities [147]. This number has undoubtedly increased and is increasing at an accelerating rate. The result is that even if the promised large improvements in development speed by the use of new methods do eventually appear they will have little impact on total software costs since any gain from increased development will be swallowed up by the increased maintenance cost [163]. Concern is growing that the development of new software is outpacing the ability to maintain it. In the current decade, four out of seven programmers are working on enhancement and repair projects. With large portions of software budgets being devoted to maintenance, few resources remain for new development. If these trends continue, eventually no resources will be left to develop new systems, and people will enter the Middle Ages of the information age, referred as "legacy crisis" [142].

The term "legacy system" describes an old system which remains in operation within an organisation [171]. Organisations are in fear of their legacy systems. They are afraid of keeping them, since maintaining them is a significant drain on the organisation's resources. They are also afraid of replacing them. A major reason is that those legacy systems are enormously valuable assets. Having stood the test of time and evolved, they provide the most accurate statement of current business practices. Legacy systems, then, are not an issue that will simply go away. Because legacy software systems are so critical to an organisation's survival, they are a very real problem facing many organisations and, thus, people need to develop an appropriate strategy for dealing with

them.

2.2 Software Evolution and Software Re-engineering

Software systems need to evolve continually to cope with ever-changing software requirements. Software evolution is a process of conducting continuous software re-engineering. In other words, software evolution is repeated software re-engineering [180]. As a combination of reverse engineering and forward engineering, software re-engineering technology is a practical solution for the problem of evolving existing software.

2.2.1 Characteristics of Software Evolution

Much attention should be paid to the following principles when a software system is evolved:

- the evolved system should be reliable,
- the evolved system should be functional,
- the evolved system should be efficient, and
- the cost of evolution should be acceptable.

The above principles must be satisfied; otherwise, it leads to high cost of a software system and sometimes, it implies the redesign of the whole system, which requires huge investment, with significant risk that the new systems may fail to deliver the required services [98].

2.2.2 Laws of Software Evolution

Lehman clarified classification scheme distinguishing three types of programs S, P and E [91, 92], and defined that an E-type program is a computer program that solves a problem or implement a computer application in the real world domain [90]. They

indicated that E-type software supports E-type applications and the latter must also evolve [91]. The results of their studies are based on observation, which have become known as Lehmans's laws (depicted in Table 2-1) [89]:

Law Name	Law Description
I. Continuing Change	An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.
II. Increasing Complexity	As a program is evolved its complexity increases unless work is done to maintain or reduce it.
III. Self Regulation	The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.
IV. Conservation of Organisational Stability	The average effective global activity rate on an evolving system is invariant over the product life time.
V. Conservation of Familiarity	During the active life of an evolving program, the content of successive releases is statistically invariant.
VI. Continuing Growth	Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
VII. Declining Quality	E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.
VIII. Feedback System	E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

Table 2-1. Lehman's Laws of Software Evolution [89]

The grand challenge is how to break these laws to prolong the life of the software systems. The first and second laws are especially interesting and will be discussed in detail.

2.2.3 Software Changes and Software Evolution Approaches

Large-scale industrial and commercial software systems usually have a long life-span,

sometimes twenty years or more. The consequent of software aging, as described in [124], is a growing inability to keep up with the market by introducing new features, reduced performance and decreased reliability. Many such applications do not remain static after their original development phase; they tend to evolve continuously during their lifetime. There are four main reasons for changing software [97, 180]:

- Perfective. These changes are made to improve the product, such as adding new user requirements, or to enhance performance, usability, or other system attributes. These types of changes are also called enhancements.

- Corrective. These changes are made to repair defects in the system.

- Adaptive. These changes are made to keep pace with changing environments, such as new operating systems, language compilers and tools, database management systems and other commercial components.

- Preventive. These changes are made to improve the future maintainability and reliability of a system. Unlike the preceding three reactive reasons for change, preventive changes proactively seek to simplify future evolution.

Every software system that is being used needs to be changed. Software is only finished when it is no longer in use. The activities of software change can be divided into three categories: maintenance, modernisation, and replacement [142].

- Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. Maintenance is an incremental and iterative process in which small changes are made to a system without major structural changes.

- Software modernisation involves more extensive changes than maintenance but conserves a significant portion of the existing system. These changes often include restructuring the system, enhancing functionality, or modifying software attributes. Software modernisation falls between the two extremes of system replacement and continued maintenance.

- Software replacement requires rebuilding the system from scratch and is resource intensive. Replacement is carried out when modernisation is not possible or cost-effective. Systems can be replaced incrementally where modernisation works as a preparatory step before beginning an incremental replacement effort.

Software re-engineering is a form of modernisation that improves capabilities and/or maintainability of a legacy system by introducing modern technologies and practices. The purpose of software modernisation/re-engineering is both to utilise existing software to take advantage of new technologies and to enable new development efforts to take advantage of reusing existing software, which has the potential to improve software productivity and quality across the entire life cycle.

2.2.4 Software Complexity and Modern Software Paradigm

In the past decades, software developers created massive, monolithic software programs that often performed a wide variety of tasks. Difficulty of designing, implementing and launching computer-based systems increases exponentially with the size of the system. However, there has been a shift from the development of massive programs containing millions of lines of code, to smaller, modular, pieces of code, where each module performs a well defined, focused task or a small set of tasks, rather than thousands of different tasks, as used to be the case with old legacy systems. The computer industry is always looking for ways to handle complexity and improve software development productivity as well as the quality and longevity of the software that it creates. Object-orientation, component-based development, patterns, and distributed computing infrastructures are examples of new approaches that have aided in this quest [46]. Of course, these modern software paradigms are also capable of dealing with change.

Since the birth of modern computers, it has been witnessed the progressive move from low-level abstractions to high-level. In terms of mainstream programming, there has been a progressive move from structured programming to object-oriented programming, and more recently to agent-oriented programming. There are several methodologies for developing software that might be suitable for large distributed applications. Object-oriented, component-based, service-oriented, and agent-based

techniques are the mainstreams at current stage. There are many requirements for a methodology, but it must support distributed development and execution, software reuse, and robustness [63]. It can be predicted that more advanced software development paradigms will be available in the future and therefore that evolution for software systems built with these new paradigms will also be needed. Modifying evolution techniques to keep up with the pace of emerging development paradigms is a grand challenge.

2.2.4.1 Object Oriented Paradigm

An object-oriented approach to the development of software was first proposed in the late 1960s. However, it took almost 20 years for object technologies to become widely used. As time passes, object-oriented technologies are replacing classical software development approaches [130]. Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer.

2.2.4.2 Component-based Paradigm

Component-based paradigm is probably one of the most significant techniques that have occurred during the last decade. Components target the large-scale composition of software, while maintaining simplicity in that composition. Software components aim to succeed in the area of software reuse. Components are neither an alternative nor competing with object-oriented programming. These are two orthogonal and complementary concepts. Software components are all about binary reuse, strict interface/implementation separation and application development by assembly, while object-oriented programming is an approach for fine-grained code development: the coding of core routines, algorithms and data structures. Object-oriented programming can be used for component development and even as glue between components. From a view of modernisation, an important property of components is that the interfaces also hide the age of the components, permitting cooperation of legacy components and newly created system parts. This also implies that there should be a strong relationship between techniques for modernisation and techniques for construction.

2.2.4.3 Service Oriented Paradigm

As defined by the World Wide Web Consortium, A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML [161]. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.

Web services are software building blocks that can be assembled to construct the next generation of distributed business applications. Web services rely on the functionalities of publish, find, bind and the components of a Web Service model include Service Providers, Service Broker and Service Requester. Web Services are defined by their interfaces in particular about how they describe their functionality, how they register their presence, and how they communicate with other Web Services. People who want to use Web Services could connect to the Universal Description Discovery and Integration (UDDI) center to search for the required services. The information about the Web Services described by Web Service Description Language (WSDL) can be acquired. And the users could use the Simple Object Access Protocol (SOAP) to transfer the requirement information and receive the real service [160].

Today, Web services are emerging as the new "standard" architectural style. This new architectural style and the software lifecycle it implies are extremely attractive because they can effectively address the demands for short development cycles, distributed development and global user base, at the same time [151]. The Service Oriented Architecture (SOA) is the most flexible approaches based on the following reasons [59]:

- Reduction of interface complexity.
- Decentralised software development.
- Explicit separation of business logic and service mediation logic.
- Technical independency of service participants.

Using Web services technologies to implement a distributed system does not magically turn distributed object architecture into SOA, nor are Web services technologies necessarily the best choice for implementing SOAs. Nevertheless, Web services are increasingly becoming an adequate technology for the partial implementation of features of a SOA.

2.2.4.4 Agent-based Paradigm

An agent is an encapsulated computer system that is situated in a certain environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives. A multi-agent system (MAS) can be defined as a loosely coupled network of entities that work together to make decisions or solve problems that are beyond the individual capabilities or knowledge of each entity [72]. Individual agents are easier to construct and understand than large monolithic systems and agent-oriented approaches can significantly enhance the ability to model, design and build complex, distributed software systems. Because of the autonomy and intelligent nature of agents, agent-oriented techniques are a design metaphor for Web services, where agents are needed both to provide services, and to make best use of the resources available. A number of efforts have recently begun to integrate the agent and Web service communities, which enable agents to use Web services' infrastructure and to extend the Web services model with the benefits of agent technology [18].

2.3 Model Driven Re-engineering

The term "model" is used in many contexts and often has different meanings. Generally, a model can mean an abstraction and representation of the important factors of a complex reality, which is different from the thing it models, yet has the same main characteristics as the original. With a model, people can build up an understanding to the point where it is natural to answer a question by modelling it into some simpler context, answering the question with the model, and then interpreting the answer [112].

A model may hide or mask details and bring out the big picture, helping people work

at a higher level of abstraction. With complex mechanical or electronic machines or large buildings, the designers and constructors have some models (e.g. blueprints and floor plans) which provide an accurate overview and geometric representation of the structure. A model plays the analogous role in software development. Creating a model of a system is a useful way to understand a system.

Modelling is an essential part of large complex software projects. Experienced application developers often invest more time in building models than they do in actually writing code. Software modelling assumes that abstract models convey information more effectively than source code. People should use abstract models to enhance their understanding of a system and to provide a common base for others to discuss it [171]. Well-constructed models make it easier to deliver large, complex systems on time and within budget.

2.3.1 Model Driven Architecture

Model Driven Architecture (MDA) is a standard produced by the Object Management Group (OMG). The core idea of MDA is to use models and modelling as the main artefacts and activities in software development, which increases the power of models. The term 'model-driven' means that it provides a way for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification [121]. The term "architecture" means a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors [121, 146]. The MDA prescribes certain kinds of models to be used, how those models may be prepared and the relationships of the different kinds of models [121].

The MDA specification puts emphasis on different level of models, including Computational Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). A CIM describes concepts related to a particular domain with no reference to the particular problem to be solved in that domain. PIM describes a particular system that solve a particular problem but in a technology independent manner, while a PSM describes how this system can be implemented using

a given technology. Fowler suggests in [44] another distinction based on 3 levels of models, namely Conceptual Models, Specification Models and Implementation Models. Conceptual Models, the more abstract ones, describe concepts rather than solutions. They are closed to CIM. Specification Models are used to specify the system to be built but without giving details about its actual implementation. Finally Implementation Models describe how systems have to be implemented. Though specification models are close to MDA PIMs and implementation models are close to implementation models, these equivalencies do not hold because the two classifications are based on a slightly different perspective. Fowler put the emphasis on the fact, that each kind of models can be described using the same modelling language [44].

Mellor and his colleague in [103] classify models as sketch model, blueprint model, or executable model. A sketchy model is not precise or complete, being used to try out an idea when the model is a specification or to simplify communication and understanding when the model is descriptive. A blueprint model is more precise and can be used as specification to build a system. Executable Models, such as Executable UML, can be directly interpreted by a processor or to derive an executable system.

A common pattern of MDA development is to define a PIM, and to apply transformations to this PIM to obtain one or more PSMs and then generate the code based on the PSMs. The main benefit of this approach stems from the automatic transformation process that may reduce development costs and improve software quality. The automation requires that models in the context of MDA should be machine-readable that can be accessed repeatedly and automatically transformed by tools into schemas, code skeletons, test harnesses, integration code, and deployment scripts for various platforms [81]. To enable automatic transformation of a model, the model written in a language must obey the following definitions [81]:

A model *is a description of (part of) a system written in a well-defined language.*

A well-defined language *is a language with formal form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer.*

Modelling languages define what models are considered to be valid. They describe

the allowed model primitives and how these primitives may be combined to form a valid model. The primitives of a model allow the representation of the different aspects and concepts of the problem domain. A formal language has a formal syntax as well as a formal semantics. A formal syntax describes the allowed models in a precise, unambiguous way. A formal semantics assigns a precise, mathematical meaning to each of the allowed models. On the other hand, models expressed in informal languages, are relatively easy to understand by offering graphical model primitives. Informal languages allow a certain degree of freedom in the use of the different model primitives without being hindered by all kinds of strict semantic rules. The terms formal and informal should not be interpreted too strictly. Languages that are supported by a compiler or simulator are more formal than languages that do not have these facilities [81].

The checking of consistency between models is terribly complicated if these models are informal. Consistency checking of complex models becomes unmanageable without the availability of advanced software tools. The development of such tools requires a considerable degree of formality. There are costs to maintaining a large number of models and keeping them synchronised with each other. Tools can reduce those costs by automating some of the synchronisation, but they don't eliminate all of the costs.

MDA provides a framework based on the Unified Modelling Language (UML) and other industry standards for visualising, storing, and exchanging software designs and models, which include the Unified Modelling Language (UML) [122], Meta-Object Facility (MOF) [118] and XML Meta-Data Interchange (XMI) [114, 115], etc. In the context of MDA, much effort has been invested in MOF, language definition and extension mechanisms (UML and UML profiles), model transformation specification (MOF Query/View/Transformation RFP [119]), and tool support. These developments constitute enabling technologies to model-driven development.

2.3.2 Model Driven Engineering

The concept of Model Driven Engineering (MDE) is emerged as a generalisation of the MDA approach for software development. In [77], MDE is defined on the base of MDA

by adding the notion of software development process and modelling space for organising models. Model-driven engineering is a subset of system engineering in which the process heavily relies on the use of models and model engineering [39].

In general, a model of a software system is an abstract representation of this system. There exists more than one system satisfying the model, namely each model implicitly defines a collection of systems called the realisation space of the model. Notice that the more properties the model allows to be determined, the smaller the realisation space will be [157].

MDE is an open and integrative approach that embraces many other Technological Spaces (TSs) in a uniform way [85]. An important aspect of MDE is the emphasis it puts on bridges between TSs, and on the integration of bodies of knowledge developed by different research communities, which is especially useful for software modelling [39]. Not all technological spaces are equivalent. The notion of technological space has been introduced in [85]:

> **A Technological Space (TS)** *is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities.*

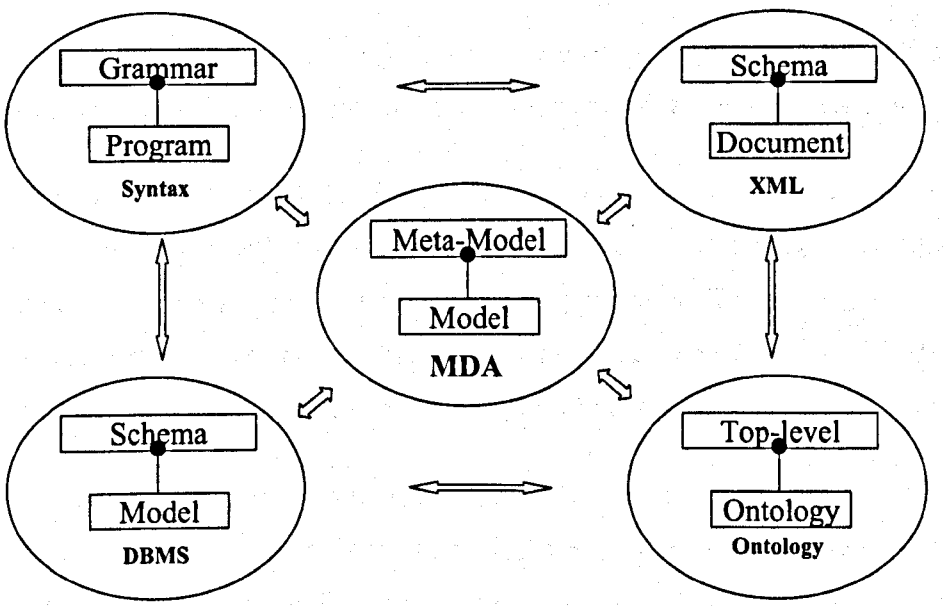


Figure 2-1. Five TSs and Their Links [85]

In [85], technological spaces are divided into Documentware, that is technology for structured document (e.g. XML), Grammarware, that is technology based on grammars, Dataware and database management systems, Ontologyware and ontology engineering, Modelware and model-based technology (e.g. UML), etc. Figure 2-1 shows relationship of above five TSs. The existence of various TSs means that given a system, one has to choose the TS that will be most appropriate for the expression of a model or a given usage.

Modelling Maturity Levels (MMLs) can relevantly indicate what role models play in software development process [170]:

- At the lowest level, the specification of the software is in the heads of the developers only and there is actually no documentation at all.
- At modelling maturity level 1, the specification of the software is written down in one or more natural language documentations.
- At modelling maturity level 2, the specification of the software is provided by one or more natural language documentations augmented with several high-level diagrams to explain the overall architecture and/or some complex details.
- At modelling maturity level 3, a set of models (diagram or text with a very specific and well-defined meaning) forms the specification of the software.
- At modelling maturity level 4, a model is a consistent and coherent set of texts and /or diagrams with a very specific and well-defined meaning. This is the first level at which a model can be understood by machine. The models at this level are precise enough to have a direct link with the actual code. Level 4 is the level at which the Model Driven Architecture (MDA) is target.
- At modelling maturity level 5, models are complete, consistent, detailed, and precise description of the system, which are good enough to enable complete code generation. This is future technology and the ultimate goal of the software modelling. At this level, source code will be invisible to the developers and

maintainer and there is no need to look into it.

2.3.3 Model Driven Reverse Engineering

Nowadays, the MDE research community and the reverse engineering community have a lot to share because models are cornerstones of both disciplines [38]. In fact Modelling and Reverse Engineering both refer to the activity of creating descriptive models. Models can be used either to specify a system to be built, or to describe an existing system. This leads to the distinction between specification models and descriptive models [143]. New systems are produced from specification models, while descriptive models are produced from existing systems.

In the context of software modelling, reverse engineering is defined as the process in which software artefacts from legacy system are restructured through model transformation based on well-defined steps. Techniques that control the changes in the model in a systematic manner are a key to model transformation, which is accomplished by specifying metamodels of well-defined transformations. The specified metamodels are used to constrain how transformations are carried out at the model level. There are also several model transformation languages available currently, such as MDR (MetaData Repository from Sun [152]), EMF (Eclipse Modelling Framework from IBM [64]) and QVT (MOF Query/ Views/ Transformations from OMG [119]).

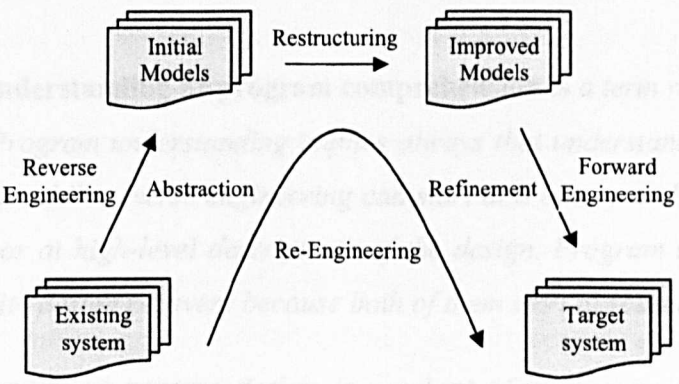


Figure 2-2. Re-engineering in Context of MDA

There are a number of techniques related to software re-engineering. All of them aim

at supporting the understanding and the reuse of assets from the previous development. Without covering all of them, the following key terms provide a clear scope and taxonomy of the domain of software re-engineering [2, 22, 142, 145, 180], which are important in the context of MDA (Figure 2-2):

Re-engineering *is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. The process of re-engineering computing systems involves three main steps: reverse engineering, restructuring, and forward engineering.*

Forward engineering *is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.*

Reverse engineering *is the process of analyzing a subject system to (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or higher level of abstraction.*

Restructuring or Refactoring *is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (i.e., functionality and semantics) yet improves its internal structure. Refactoring makes reuse of both the domain knowledge and the source code.*

Program understanding or program comprehension *is a term related to reverse engineering. Program understanding implies always that understanding begins with the source code while reverse engineering can start at a binary and executable form of the system or at high-level descriptions of the design. Program understanding is comparable with design recovery because both of them start at source code level.*

Design recovery or reverse design *is a subset of reverse engineering. Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains.*

Program Transformation *is the act of changing one program into another. The term program transformation is also used for a formal description of an algorithm that implements program transformation. The languages in which the program being transformed and the resulting program are written are called the source and target languages, respectively.*

Model Transformation *is a mapping of a set of models onto another set of models or onto themselves, which can be broken into two broad categories: model translation and model rephrasing. In the former, a model is transformed into a model of a different language, and in the latter, a model is changed in same modelling language.*

2.4 Summary

In this chapter, the background and basic concepts of model driven software modernisation are introduced.

- Many methods and techniques have been hailed as the solution to the software crisis, however in practice only small gains in productivity have been achieved. The development of new software is outpacing the ability to maintain it, referred as "legacy crisis". There is no single best approach to a solution for the legacy crisis.

- Legacy systems are operational systems which have been developed according to dated practice and technology. The major challenge is to align changing business goals and changing technologies, while preserving the assets that are hidden in the legacy systems supporting today's business operations.

- Change and Complexity are inherent properties of software systems. Software systems continue to grow in complexity at a rapid pace, resulting in systems that are increasingly complex to build and evolve to meet changing requirements.

- Software re-engineering/modernisation is a means for transforming a legacy system to its evolutionary complement, which offers several benefits not shared by

maintenance or system replacement.

- The researches on model driven software modernisation are related with re-engineering methodologies, technologies, tools and managements. The model driven approach will promote the software evolution smoothly and effectively.

Chapter 3

Related Research

Objectives

- To review the state of the art of software re-engineering approaches.
- To introduce Wide Spectrum Language and program transformation theory.
- To discuss software architecture and software evolution.
- To discuss feature modelling techniques and software evolution.
- To discuss the work related to UML.
- To review three projects related to model driven software modernisation.

3.1 Two Approaches in Software Re-engineering

3.1.1 Formal Methods and Software Re-engineering

Formal methods can be defined as mathematically based languages, techniques, and tools for specifying and verifying systems. Baumann [8] states that reverse engineering methods must be based on a sound foundation, which entails formal denotation semantics, because if these methods should extract the wrong information during reverse engineering process, this wrong information could lead to new errors in the re-engineered programs. Formal methods can also increase the understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might go undetected [25]. In the area of reverse engineering, formal methods have been put

forward as a means to

- formally specify and verify existing systems in particular those already operating in safety-critical applications,
- introduce new functionalities, and/or
- take advantage of the improvement in systems design techniques [97].

Formal methods can be classified into the following five classes or types, i.e., Model-based, Logic-based, Algebraic, Process Algebra and Net-based (Graphical) methods, which should consist of some essential components: a semantic model, a specification language (notation), a verification system/refinement calculus, development guidelines and supporting tools [180]:

- The semantic model is a sound mathematical/logical structure within which all terms, formulas and rules used have a precise meaning. The semantic model should reflect the underlying computational model of the intended application.
- The specification language is a set of notations which are used to describe the intended behaviour of the system. This language must have a proper semantics within the semantic model.
- Verification system/refinement calculi are sound rules that allow the verification of properties and/or the refinement of specifications.
- Development Guidelines are steps showing the use of the method.
- Supporting tools involve proof assistant, syntax and type checker, animator, and prototype.

There are at least two advantages of using formal methods as the foundation of software reengineering. First, formal methods can help software engineers to acquire a rigorous and precise description of the system being reengineered, therefore greatly increasing the quality of the new system. Second, automation is one of the key goals of

reengineering. By applying formal methods, it is possible to automate more of the process of reengineering [180].

3.1.2 Cognitive Approach to Software Re-engineering

Compared to formal methods, cognitive methods rely mainly on domain knowledge. In order to jump from one level up to another abstract level in the process of reverse engineering. One has to throw away some information. No method can guarantee that such a throwing away of information is appropriate [95]. This implies that the abstraction is creative work. In order to achieve correct and practical abstraction, a knowledge base is necessary.

A cognitive model describes the mental process or faculty of knowing a software system [131]. A hierarchy of cognitive design elements to support the construction of a mental model was defined in [150], which explains how to improve program understanding by supporting the actions of identifying software artifacts and the relations between them, by browsing code in delocalised plans, and by building abstractions. These actions comprise canonical reverse-engineering activities.

Two common approaches to program understanding are a functional approach emphasising cognition by what a system does and a behavioural approach emphasising how a system performs [131].

- The functional approach is bottom up and deductive, relying more on the knowledge of the implementation domain to produce higher level of abstractions that may map to the application domain and the system's functional requirements.
- The behavioural approach is top down and inductive, using hypothesis postulation and refinement to match artifacts derived from knowledge of the application domain onto the related software system.

3.2 Wide Spectrum Language (WSL)

Wide Spectrum Language (WSL) [87, 165, 167-169, 180] has been developed for a number of years and has been used to build a general approach and a tool for addressing research issues such as program comprehension and reverse engineering using program transformation and abstraction techniques. WSL is built on formal methods and supports both object oriented and structural elements of software systems. In order to be successful in reverse engineering, WSL meets at least three conditions:

- WSL must be applicable to parts of the programming language that is relevant for reverse engineering methods.
- WSL should support standard approaches to program analysis such as control-flow analysis, data-flow analysis, etc.
- WSL should be easily and efficiently implementable.

WSL was developed with several advantages in mind:

- The ability to express general specifications in terms of mathematical logic with suitable notation.
- A well-developed library of proven transformations that do not require the user to fulfil complex proof obligations before these transformations can be applied.
- Techniques to bridge the "abstraction gap" between specifications and programs.
- The ability to scale to large programs and applicability to real programs.

The WSL language is built up in a series of stages or levels, starting with a very small and mathematically tractable kernel language. The "Spectrum" in "Re-engineering Wide Spectrum Language" refers to the range of operations, from "low level" things, such as program structures and commands, to high level operations, such as specification statements. By translating a legacy system's source code to WSL as an intermediate representation, it allows the re-engineering effort to be divided up into smaller steps

rather than as a monolithic source to target domain re-engineering effort [108].

3.2.1 WSL Kernel Language

The WSL kernel language is based on infinitary first order logic, which is an extension of ordinary first order logic which allows conjunction and disjunction over (countably) infinite lists of formulae, and quantification over finite lists of variables.

Expressions and conditions (formulae) in WSL are taken directly from infinitary first order logic. Statements in the kernel language are constructed by combining infinitary logic formulae, lists of variables and statement variables. Four primitive statements and three compound statements are needed to define the whole kernel language. Let P and Q be any infinitary logical formulae and x and y be any finite, non-empty lists of variables. The primitive statements are [169, 180]:

- **Assertion**: *{P}* is an assertion statement which acts as a partial skip statement. If the formula P is true then the statement terminates immediately without changing any variables, otherwise it aborts (Abnormal termination and non-termination are treated as equivalent, so a program which aborts is equivalent to one which never terminates);

- **Guard**: *[Q]* is a guard statement. It always terminates, and enforces Q to be true at this point in the program without changing the values of any variables. It has the effect of restricting previous nondeterminism to those cases which will cause Q to be true at this point. If this cannot be ensured then the set of possible final states is empty, and therefore all the final states will satisfy any desired condition (including Q);

- **Add variables**: *add(x)* first ensures that the variables in x are in the state space (by adding them if necessary) and then assigns arbitrary values to the variables in x. The arbitrary values may be restricted to particular values by a subsequent guard;

- **Remove variables**: *remove(y)* ensures that the variables in y are not present in

the state space (by removing them if necessary).

The compound statements are:

- **Sequence**: *(S1; S2)* executes *S1* followed by *S2*;
- **Nondeterministic choice**: *(S1 Π S2)* chooses one of *S1* or *S2* for execution, the choice being made nondeterministically;
- **Recursion**: *(μX.S1)* where *X* is a statement variable (a symbol taken from a suitable set of symbols). The statement *S1* may contain occurrences of *X* as one or more of its component statements. These represent recursive calls to the procedure whose body is *S1*.

3.2.2 Semantics of Kernel Language

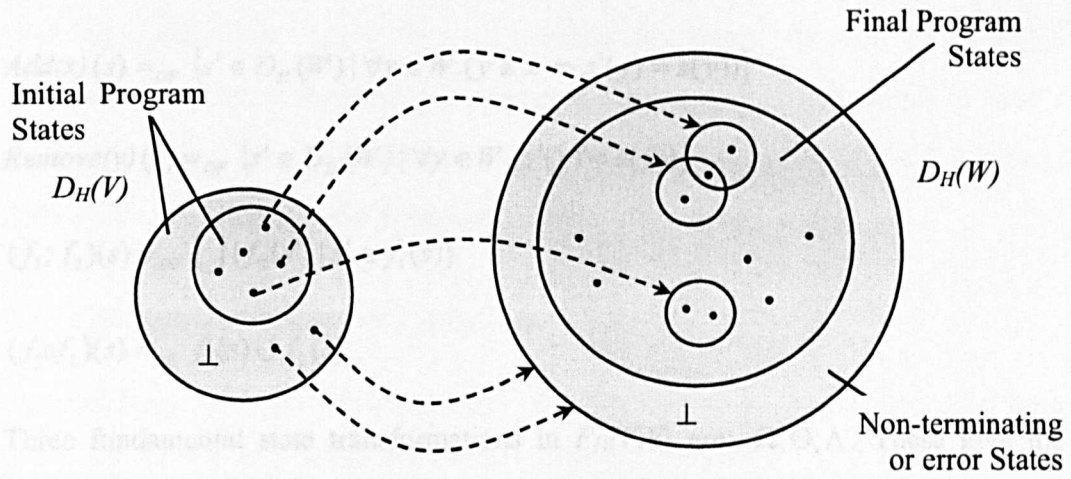


Figure 3-1. Semantics of Program State Transformation

The state transformation is illustrated in Figure 3-1. Let *V* and *W* be finite sets of variables and *H* be a set of values. A state is either the special ⊥ which indicates nontermination or error, or is function from *V* to *H*. This function gives a value (taken from *H*) to each variable in the state space. The set of all state on *V* is denoted $D_H(V)$ where $D_H(V) =_{DF} \{\perp\} \cup H^V$. A state predicate is a set of proper states (i.e. states other

than $\perp$), with the set of all state predicates denoted $E_H(V)$. So $E_H(V) =_{DF} \mathbb{P}(H^V)$. A *state transformation* is a function which maps a state in V^H to a set of states in W^H, where $\perp$ maps to W^H and if $\perp$ is the output, then so is every other state [87, 169].

The set of all state transformation from V to W may therefore be defined as:

$$F_H(V,W) =_{DF} \{f : D_H(V) \rightarrow (E_H(W) \cup (D_H(W))) \mid f(\perp) = D_H(W)\}$$

The non-recursive kernel language statements are defined as state transformations as follows:

$$\{e\}(s) =_{DF} \begin{cases} \{s\} & \text{if } s \in e \\ D_H(W) & \text{otherwise} \end{cases}$$

$$[e](s) =_{DF} \begin{cases} \{s\} & \text{if } s \in e \\ \phi & \text{otherwise} \end{cases}$$

$$Add(x)(s) =_{DF} \{s' \in D_H(W) \mid \forall y \in W.(y \notin x \Rightarrow s'(y) = s(y))\}$$

$$Remove(y)(s) =_{DF} \{s' \in D_H(W) \mid \forall y \in W.(s'(y) = s(y))\}$$

$$(f_1; f_2)(s) =_{DF} \bigcup \{f_2(s') \mid s' \in f_1(s)\}$$

$$(f_1 \sqcap f_2)(s) =_{DF} f_1(s) \cup f_2(s)$$

Three fundamental state transformations in $F_H(V,V)$ are: Ω, Θ, Λ. These give the semantics of the statements abort, null and skip, where abort is defined as *{false}*, *null* is defined as *[false]* and skip is defined as *{true}*. For each proper $s \in D_H$:

$$\Omega(s) =_{DF} D_H(V) \qquad \Theta(s) =_{DF} \phi \qquad \Lambda(s) =_{DF} \{s\}$$

Recursion is defined in terms of a function on state transformations:

Definition 3.1 *Recursion*: Suppose a function F which maps the set of state transformations $F_H(V,V)$ to itself. A recursive state transformation from F as the limit

of the sequence of state transformations $F(\Omega), F(F(\Omega)), F(F(F(\Omega))), \cdots$ with the definition of state transformation given above, this limit $(\mu.F)$ has a particularly simple and elegant definition:

$$(\mu.F) =_{DF} \coprod_{n<\omega} F^n(\Omega)$$

where $\coprod$ on a set of state transformation is defined by pointwise intersection:

$$(\coprod X)(s) =_{DF} \bigcap \{f(s) \mid f \in X\}$$

$F^n(\Omega)$ is the "nth truncation" of $(\mu.F)$: as n increases the truncations get closer to $(\mu.F)$. The later truncations provide more information about $(\mu.F)$ - more initial states for which it terminates and a restricted set of final states. The $\coprod$ operation collects together all this information to form $(\mu.F)$.

With this definition, $(\mu.F)$ is well defined for every function F: $F_H(V,V) \rightarrow F_H(V,V)$. However if the recursive statement state transformations needs to satisfy the property $F((\mu.F)) = (\mu.F)$ (in other words, to be a fixed point of the F function) then the further restrictions on F is required [169].

3.2.3 Extensions to Kernel Language

The kernel language is particularly elegant and tractable but is too primitive to form a useful WSL for the transformational development of programs. For this purpose it is needed to extend the language by defining new constructs in terms of the existing ones using definitional transformations, which consists of the following constructs [180]: Sequential composition; Deterministic choice; Specification statement; Simple assignment; Nondeterministic choice; Deterministic iteration; Nondeterministic iteration; Initialised local variables; Counted iteration and Block with procedure calls.

A series of new language levels is built up, with the language at each level being defined in terms of the previous level. Each new language level automatically inherits

the transformations proved at the previous level, which form the basis of a new transformation catalogue. This technique has proved extremely powerful and has led to the development of a practical transformation system that implements a large number of program transformations.

3.2.4 Program Transformation Theory

A state transformation is either a specification of a program, or a (partial) description of the behaviour of a program. If f is a specification, then for each initial state s, $f(s)$ is the allowed set of final states. If $\perp \in f(s)$ then the specification does not restrict the program in any way for initial state s, since every other state is also in $f(s)$. Similarly, if f is a program description, then $\perp \notin f(s)$ means that the program is guaranteed to terminate in some state in $f(s)$ when started in state s [169].

Definition 3.2 *Refinement*: Given two state transformations f_1 and f_2 in $F_H(V,W)$, f_2 refines f_1, or f_1 is refined by f_2, written as $f_1 \leq f_2$ if and only if f_2 satisfies f_1.

More formally:

$$f_1 \leq f_2 \Leftrightarrow \forall s \in D_H(V).f_2(s) \subseteq f_1(s)$$

If all the constant symbols, function symbols and relation symbols in the statement are interpreted as elements of H, functions on H and relations on H, then formulae can be interpreted as state predicates and statements as state transformations.

Definition 3.3 *Satisfaction*: Program f satisfies specification g precisely when $\forall s.(f(s) \subseteq g(s))$.

A program f_2 is a refinement of program f_1 if f_2 satisfies every specification satisfied by f_1, i.e. $\forall g.(\forall s.(f_1(s) \subseteq g(s)) \Rightarrow \forall s.(f_2(s) \subseteq g(s)))$. It is easy to see that refinement and satisfaction, as defined above, are identical relations.

Definition 3.4 *Equivalent*: Two statements f_1 and f_2 are equivalent if their

interpretations are identical.

The transformation rules are proved in the context of a set Δ of assumptions. Here, Δ is a finite or countably infinite set of sentences (formulae with no free variables). In any interpretation, a sentence must either be universally true or universally false. An interpretation within which all the sentences of Δ are true is called a model for Δ. If S is a statement and V and W are state transformation defined by applying M to S on V and W is denoted $\text{int}_M(S,V,W)$.

Definition 3.5 *Semantic Refinement*: Let S_1 and S_2 be statements and V and W be state spaces such that $S_1 : V \to W$ and $S_2 : V \to W$. Let Δ be a set of sentences. If for every model M of Δ, if $\text{int}_M(S_1,V,W) \leq \text{int}_M(S_2,V,W)$ then S_2 is a refinement of S_1 under Δ and is written as:

$$\Delta \models S_1 \leq S_2$$

If $\Delta \models S_1 \leq S_2$ and $\Delta \models S_2 \leq S_1$ then the semantic functions are identical under every model, so S_1 and S_2 are semantically equivalent and is written as

$$\Delta \models S_1 \approx S_2$$

3.2.5 *Meta*WSL for Program Transformation

A transformation is a function which maps a WSL program to an equivalent WSL program. WSL programs are represented as abstract syntax trees: therefore a transformation can be expressed as an operation on a syntax tree. Similarly, the applicability condition of a transformation can be expressed as a function on syntax trees. By extending the WSL language to provide suitable constructs for accessing and manipulating WSL syntax trees, transformations can be expressed in this extension of WSL, called *Meta*WSL [168]. Since *Meta*WSL is also an extension to WSL, the WSL transformations can also be applied to *Meta*WSL code itself that further *Meta*WSL specific transformations are possible.

A program transformation can be implemented as a piece of *Meta*WSL code which in turn can be the source program for applying a transformation (including itself: a transformation can be applied to its own source code). The result will be a different implementation of the same program transformation. This "reflexivity" in the system has several advantages: the correctness of the implementation of a transformation can be proved by transforming a specification of the transformation into an implementation of the transformation with proven transformations and the efficiency of the transformation system can be improved by transforming the source code into a more efficient implementation.

3.2.6 Extensions to Object Orientation

In order to support object oriented technology, the WSL was extended to include such constructs as class structure which contains both variables (attributes) and procedures (methods or operations) [97, 105].

The syntax of extended WSL adds the following object-oriented portion:

1. Class Definition

```
Class T
  Var
    Ti : xi;   /* Attributes of Class */
  Proc
    mj(In pinjk:Tk, Out poutjl:Tl)   /* Methods of Class */
      Begin
        Aj; /* WSL statements */
      End
  End
```

This statement is the class building declaration. It defines a class named T, which has data fields x_i of type T_i and methods m_j. pin_{jk} stands for the input parameters of method m_j, and $pout_{jl}$ stands for the output parameters of method m_j.

2. Class Hierarchy

```
T Extends T'
```

This statement is used to build the object hierarchy. It declares that class T is a subclass of class T'. Therefore, T inherits the properties of T'.

3. Field Reference

x.d

This is object field reference. x is an object and d is a field of x.

4. Method Invocation

x.m (In ek, Out yl)

This invokes the method m in object x.

5. Object Declaration

T : x

This statement defines x as a variable of type T. If T is a class, x will be an object of class T.

3.2.7 WSL Related Tools

3.2.7.1 Maintainer's Assistant

One of the most important successes of Maintainer's Assistant (MA) [13, 176, 180] is that it is based on a wide spectrum language, which defines syntax and semantics formally. Maintainer's Assistant (MA) employs transformation techniques to derive a specification from a section of code and to transform a section of code into a logically equivalent form. MA has features as follows:

- It acts, initially, on existing program code as a tool to aid comprehension (possibly by producing specifications) and only the program code is required for the processing;
- The system can work with any language by first translating, i.e., with a standalone translator into WSL and changes are made to the WSL program by means of transformation;
- The system incorporates a large, flexible catalogue of transformations. The applicability of each transformation is tested before it can be applied;

- The system is interactive and incorporates an X-Windows front end and pretty printer called the Browser;
- The system includes a database structure to store information about the program being transformed, such as the variables assigned to within a given piece of code;
- The system includes a facility to calculate metrics for the code being transformed.

3.2.7.2 FermaT

Maintainer's Assistant has evolved into an industrial-strength re-engineering tool, FermaT [165, 166, 180], which allows transformations and code simplification to be carried out automatically. The FermaT tool was also designed to use WSL and has applications in the following areas:

- Improving the maintainability of existing mission-critical software.
- Translating programs into modern programming languages. FermaT often translates program written in obsolete assembler language to more modern languages such as C.
- Extracting reusable components from the current system, deriving their specifications, and storing the specifications, implementation, and development strategy.
- Reverse engineering existing systems to high-level specifications, followed by subsequent re-engineering and evolutionary development.

3.2.7.3 TAGDUR

TAGDUR (Transformation and Automatic Generation of Documentation in UML through Re-engineering) [105-107], was designed to overcome the lack of documentation problem often faced by legacy systems whose original documentation has been lost. By utilising information acquired during the transformational process and

by parsing the code of the transformed system, this tool generates UML diagrams of the transformed system.

3.3 Architecture-based Software Evolution

3.3.1 Software Architecture

As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organisation of a system as a collection of components; global control structures; the protocols for communication, synchronisation and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; selection among design alternatives and non-functional properties. This is the software architecture level of design [146]. The focus of architecture-based software development is shifted from lines-of-code to coarser-grained building blocks and their overall interconnection structure. Software architecture can be summarised as follows [7, 49, 101, 184]:

- Software architecture deals with the design and implementation of the high-level structure of the overall software system. The software architecture of a system is an artefact that is the result of the software design activity.

- Software architecture is a description of subsystems, components of a software system and the relationships between them. Components and connectors are recognised as the fundamental ingredients of software architecture.

A number of the basic ingredients of architectural description can be identified [6, 10, 58, 62, 84], Components, Connectors, Configurations for an overall architecture. Architecture Description Language (ADL) is defined in [146] with six properties:

- Composition and decomposition: The former is the process of integration of system components into larger sub-systems, while the latter the process of

decomposition of a system into its constituents.

- Abstraction: A system has abstract views of high-level or low-level design.
- Reusability: Generic patterns of components and connectors are defined.
- Configurability: The structure of software systems can be changed independently from the components.
- Heterogeneity: Various architectural styles or programming languages can be accommodated in one system.
- Analysis: Architectural properties, metrics or simulating run-time characteristics are provided for analysing the system.

3.3.1.1 Software Patterns

Patterns help build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practice. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Patterns can be used to construct software architecture with specific properties. There are several properties of patterns [14]:

- A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.
- Patterns document existing, well-proven design experience.
- Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.
- Patterns provide a common vocabulary and understanding for design principles.
- Patterns are a means of documenting software architectures.

- Patterns support the construction of software with defined properties.
- Patterns help to build complex and heterogeneous software architectures.
- Patterns help to manage software complexity.

Patterns occur at different levels: Architectural Patterns; Design Patterns; Idioms. Patterns make an important contribution to the benefits that can gain from software architecture [184]:

- They help with the recognition of common paradigms, so that high-level relationships between software systems can be understood and new applications built as variations on old systems.
- They stress the importance of non-functional properties, such as changeability and reliability.
- They provide support for finding an appropriate architecture for the software system under development.
- They provide support for making principled choices among design alternatives.
- They help with the analysis and description of high-level properties of complex software systems.
- They provide support for change and evolution of software systems.

3.3.1.2 Aspect Oriented Programming (AOP)

Software system is often required to add new general functions, which are distributed into many components of the system. There are many disadvantages if these functions are inserted into every needed place directly: Firstly, it is too complex to do so. Secondly, it may dramatically increase the risk of introducing errors into the software system. If k out of m modules are modified, the number of module interface checks required, N, is N = (k*(m-k) + k*(k-1))/2 [52, 127], which means that more testing

needs to be done. Further more, such an approach may destroy the structure and the encapsulation of the system, which will lead to 'tangled' code. By introducing the AOP and deploying 'Joinpoints', the proposed approach can insert new code into the evolving system without any modifications to the existing class structures. In 1997, a new programming methodology, Aspect-Oriented Programming (AOP) [80], was proposed. AOP has been introduced as a technique for separation of concerns that copes with scattering and tangling of non-functional code over multiple classes. Most AOP researches focus on providing aspect-oriented implementations of object-oriented design patterns and it is implemented with a few programming languages, which support aspect-oriented functions [3, 26, 60, 113].

The core concept in AOP is the Joinpoint, which was first mentioned in AspectJ and is a well-defined point in the execution of a program-like method calls, loop beginnings and object constructions [159]. The common behaviour, which crosscuts many components, is named as crosscutting concern. AOP enables programmer to modularise common behaviours and encapsulate them in a new component [45, 80], which can be coded and revised independently and be injected into the existing component code with a 'weaver' [79]. This kind of injection can be either static or dynamic. The static weaving is the most widespread method, in which the application is modified by adding the new functionality in the form of Aspects and recompiling it. Dynamic weaving is runtime weaving. It can weave Aspects at runtime while the application is operational and without interrupting its operations [41, 86]. Both static and dynamic approaches make the system structure being simple and clear.

The task of AOP-based software evolution involves both the analysis of the source code and the injection of new functions. AOP provides some mechanisms (Joinpoints, Advice, and Aspect Weaving) that allow modifying the behaviour and the structure of an application, also of a non-stopping application [129]. AOP supports evolution via crosscuts that are sets of events (method calls, exception raises, etc.) to be intercepted, and Advice that is to be executed when these events are activated. The insertion of Advice is accomplished through static code transformation (evocatively called 'weaving'). Crosscuts and Advice are integrated into a static scoping device called an

Aspect that allows AOP programmers to conceptualise and integrate otherwise scattered changes to a system. Both the Advice and the crosscuts are language-specific mechanisms [32]. Currently, studies on AOP-based software evolution focus on dynamic evolution in distributed and heterogeneous system. Devanbu and Wohlstadter have proposed a multi-tiered, eclectic approach to solve the evolution in a distributed and heterogeneous system [27, 32]. There are also some AOP-based systems, which are implemented with non AOP-based languages, such as Microsoft .NET, etc. Most of them focus on the construction of Weaver in AOP [50, 141].

3.3.2 Software Architecture Reconstruction

Software Architecture Reconstruction method [82, 83] is the process where the as-built architecture of an implemented system is obtained from an existing legacy system. This is done through a detailed system analysis and can be effectively supported by the integration of existing tools and techniques into a workbench. These tools extract information about the system and aid in building successive levels of abstraction. It consists of such key elements as Relation Partition Algebra, Architectural Views, Reconstruction Levels, InfoPacks and ArchiSpects:

- Relation Partition Algebra: *Relation Partition Algebra* has been defined to formalise descriptions of software architectures based on sets and binary relations. *Relation Partition Algebra* offers abilities to express queries for structures in a formal notation, which can be executed on the model of a software system.

- Architectural Views: *Architectural Views* are classified as logical view, module view, code view, physical view and execution view.

- Reconstruction Levels: A range of architectural aspects must be reconstructed. Software architecture *Reconstruction Levels* consist of initial level, described level, redefined level, managed level and optimised level.

- InfoPacks and ArchiSpects: An *InfoPack* is a package of particular information extracted from the source code, design documents or any other information

source. An *ArchiSpect* is a view on the system that makes explicit a certain architectural structure. It has a higher level of abstraction than an *InfoPack*. Most *ArchiSpects* build upon the results of *InfoPacks*. A complete set of *ArchiSpects* construct a system's actual architecture.

Architecture reconstruction is an iterative and interactive process, comprising four phases [75, 76, 149].

- Information extraction: A set of views/models that represent the system's fundamental structural and behavioural elements are extracted from implementation artifacts and stored in the Repository.
- View fusion: Fused views/models that augment or improve the extracted views are created.
- Reconstruction phase: Patterns are applied to the fused views/models to reconstruct architecture-level derived views/models.
- Architecture analysis: Resulting architecture is analysed.

Design patterns in software design help developers achieve high quality architectures. Patterns provide the medium for an analyst to express their understanding of a system's architecture as structural and attribute-based relationships among its components. Also, the derived views/models may be explored for the purposes of evaluating architectural conformance, identifying targets for re-engineering or reuse and analysing the architecture's qualities. Reconstructing architectures of systems designed and developed with design patterns has traditionally been approached via manual source code inspections [140]. In [56], a semi-automatic architecture recovery method based on recognised instances of design patterns is presented. As an iterative and interpretive process, it incorporates human involvement as an integral part for evaluating the results and determining which patterns to apply in different iterations.

3.3.3 ADL Tool: AcmeStudio

AcmeStudio [1] has been undertaken by the ABLE group at Carnegie Mellon University, and Dave Wile at USC's Information Sciences Institute, which is a customisable editing environment and visualisation tool for software architectural designs based on the Acme ADL. AcmeStudio is an adaptable front-end that could be used in a variety of modelling and analysis applications. AcmeStudio is implemented as a plugin for Eclipse environment, allowing easy extensions of AcmeStudio with new analyses and functionality, and customisation of new architectural environments tailored to a particular organisation. AcmeStudio is available as a free download, which has the following features [1]:

- Graphical editor for architectural designs.
- Edit designs in existing families (styles), or create new families and types.
- Create new diagram styles based on visualisation conventions people define.
- Integrated Armani constraint checker to check architectural design rules.
- Implemented as Eclipse plugin for portability and extensibility.
- Available for Windows, Linux, and MacOS X.

3.4 Feature-based Software Evolution

3.4.1 Domain Engineering

A domain, as defined by [29], is an area of knowledge, which includes the knowledge of how to build software systems or parts of software systems in that area. Domains can be vertical or horizontal, depending on the classification criteria. Vertical domains are areas organised around classes of systems, such as order processing systems, inventory management systems and payment systems. Horizontal domains are areas organised around classes of parts of systems. These parts of systems are classified according to

their functionality, such as database systems, GUI (Graphic User Interface) libraries and workflow systems.

The systematic approach for achieving the goals of using the domain knowledge is called Domain Engineering. Domain Engineering is the activity of collecting, organising, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, etc.) when building new systems [29].

Domain engineering does not serve only the building of new systems; it also enables systematic use and reuse of the domain knowledge in order to support the establishment, maintenance and evolution of software systems. Through capturing a well structured overview of the domain knowledge can contribute significantly when a software system is subject to reverse engineering [126]. The acquired domain knowledge in the form of reusable assets helps organisation understanding and overcoming the complexity of the recovered information about the systems and hence supports the architecture recovery, which is a prerequisite of successful evolution.

Domain Engineering encompasses three main processes: Domain Analysis, Domain Design, and Domain Implementation [29]. The tasks of these processes are briefly described in the list below:

- Domain Analysis aims at identifying and defining a set of reusable assets for the systems in a domain.
- Domain Design aims at establishing a common architecture for the systems in a domain.
- Domain Implementation aims at implementing the reusable assets, e.g. reusable components, domain-specific languages, generators, and a reuse infrastructure.

There exist many different domain models such as domain feature model, domain functional model, domain dynamic model, domain object model, domain information

model, domain data dictionary, etc. However, feature modelling is considered as the greatest contribution of domain engineering to software engineering [29].

3.4.2 Feature and Feature Model

There are many definitions of features, depending heavily on their context and their use [156]. The use of "features" is motivated by the fact that customers and engineers often speak of product characteristics in terms of "features the product has and/or delivers". They communicate requirements in terms of features and, to them, features are distinctively identifiable functional abstractions that must be implemented, tested, delivered and maintained.

According to [88], a feature is a prominent or distinctive aspect, quality or characteristic of a software system or systems. The Feature Oriented Domain Analysis (FODA) method [73] defines feature as "an end-user visible characteristic of a system". There are also many other definitions of features, but there is unfortunately no unified understanding what the features are. This is probably due to the fact that the domain modelling concepts have been developed in quite different ways [126].

The notion of features in this thesis is more general so that it can capture a specific set of properties in re-engineering process. For the proposed research, the more practical definition of a feature is used as a coherent and identifiable bundle of system functionality that is visible to the user via the user interface [35, 156]. However, the description of this feature definition is still not explicit enough since a feature is a functionality of a system essentially but not all the functionalities are features. To determine if a functionality of a system is a feature, the following rules can be used as the criteria for this purpose [16]:

- If the functionality can be used to specify one of the capabilities of the system;
- If the functionality can be visible or identifiable in the perspective of the end user;
- If the functionality is an instance of a domain feature;

- The granularity of the functionality should be coarse-grained without addressing the computational detail.
- If the functionality is identified as a feature and indecomposable based on the user's perspective, then the feature is called as an atomic feature.
- Two or more atomic features can consist in a composite feature based on the business rules.

When a feature is considered, it must coexist with the other features in a common feature model by their relationships. Feature model [29, 73, 88] has been explored as a base for feature engineering in different practical areas. A feature model is the result of a combined process as identifying features, classifying features, organising feature as a set of coherent models and validating the models [9]. A feature model gives a hierarchical structure to the features, where the features are structured by relations. Feature models are capable of presenting domain concepts in a structured way and hence, feature models are capable of bridging the abstraction gap between requirements and architecture.

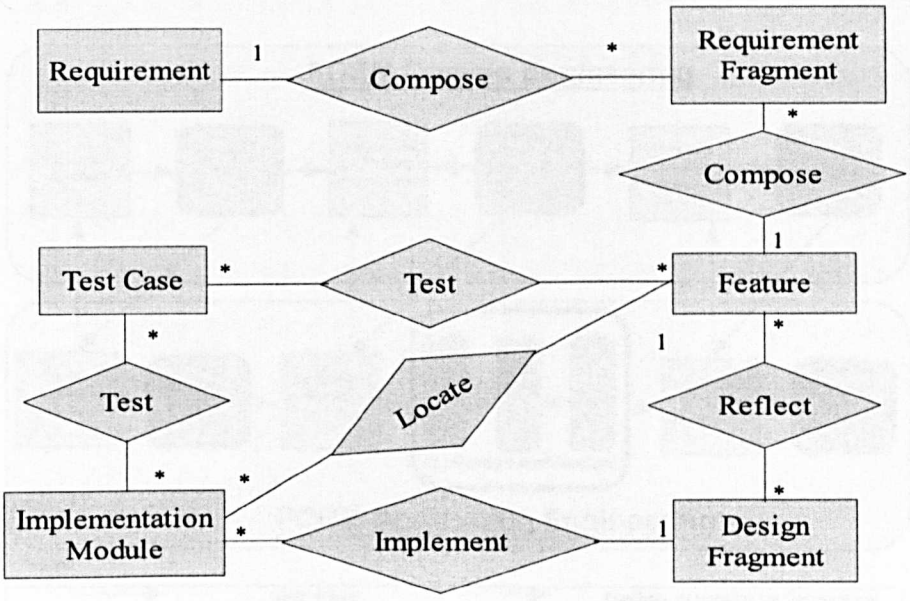


Figure 3-2. ER Diagram of Feature-Oriented Artifacts

Feature and feature model are a bridge to understand different software artifacts. It is necessary to understand and utilise feature by linking features to other entities across software lifecycle [135]. Figure 3-2 is a feature-oriented artifacts Entity-Relationship (ER) diagram [156]. The ER diagram specifies the traceable relationship between feature and the other artefacts. Feature traceability links discussed by Riebisch [136] support the mapping relationship among different kinds of artifacts. Since feature model is used as a bridge between the different levels of abstraction, it is not only a domain analysis means but also a flexible approach to refining requirement to implementation. Feature models are the means to structure the features and to express the relations between them, while feature modelling is the activity of creating the feature models.

3.4.3 Feature Modelling

To construct a feature model, a wide variety of sources need to be used to gather sufficient domain information. Some of these sources include existing systems in the domain, domain experts, textbooks, prototyping, standards, technology forecasts, etc. [29]. FORM (Feature-Oriented Reuse Method) [74] and FODA [73] are known for the introduction of feature models. Figure 3-3 shows FORM engineering processes.

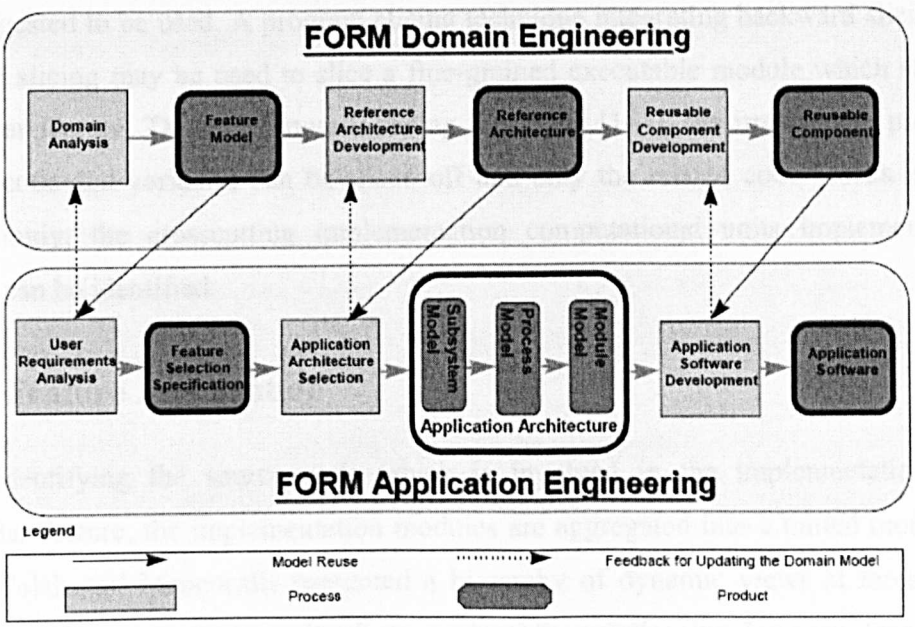


Figure 3-3. FORM Engineering Processes [74]

Feature modelling is a technology that has to be used to address the understanding of features in software systems and defines mechanisms for carrying a feature from the problem domain into the solution domain. As a result, feature models are capable of making a comprehensive overall system presentation and representing the highest elements of abstraction by which a system can be decomposed in the solution domain.

3.4.4 Feature Location

To discover feature implementation, feature location is a re-engineering technology used to locate a particular feature in the most relevant code, understand it and make the change so as to minimise unwanted side effects [156]. In some cases, a particular feature can be implemented as a piece of code that is mostly localised to a single module; but in many other cases, features cut across multiple components [54]. Through feature location, the relationship between implementation module and a particular feature can be recovered.

Many researchers have studied dynamical and static approaches [20, 35, 172] which suggest different way to locating features in their implementation modules. In order to locate feature into its implementation, the test-case based location techniques [104, 172] are suggested to be used. A program slicing technique integrating backward slicing and forward slicing may be used to slice a fine-grained executable module which serves a particular feature. Through program slicing technique [167], the irrespective pieces of source code and variables can be sliced off and only the related code blocks are left. Accordingly, the crosscutting implementation computational units implementing a feature can be identified.

3.4.5 Feature Aggregation

After identifying the source code which is involved in the implementation of a particular feature, the implementation modules are aggregated into a united module. In [139], Salah and Mancoridis presented a hierarchy of dynamic views at three levels covering different abstraction levels to analyse the relationship between feature and feature implementation such as classes and objects. Metha [104] proposed four

interactions between the identified implementation modules to construct feature-oriented components. The aggregation rules proposed in [104] can be used at this stage.

3.5 UML and Executable UML

3.5.1 Unified Modelling Language (UML)

UML is a de facto standard for representing design in a graphical way. Although UML provides a set of notations, such as Class diagram, for representing object oriented concepts, its use is not limited to object oriented systems. UML profiles (library extensions) enables customisation of the canonical core of UML syntax to represent systems in specific domains. UML has built-in mechanisms for assisting automated software development with Object Constraint Language (OCL) and Action Semantics.

UML has different types of diagrams: Use Case, Class, Object, Sequence, Collaboration, Statecharts, Activity, Component and Deployment [103], which provides multiple perspectives of the system, shown as Figure 3-4.

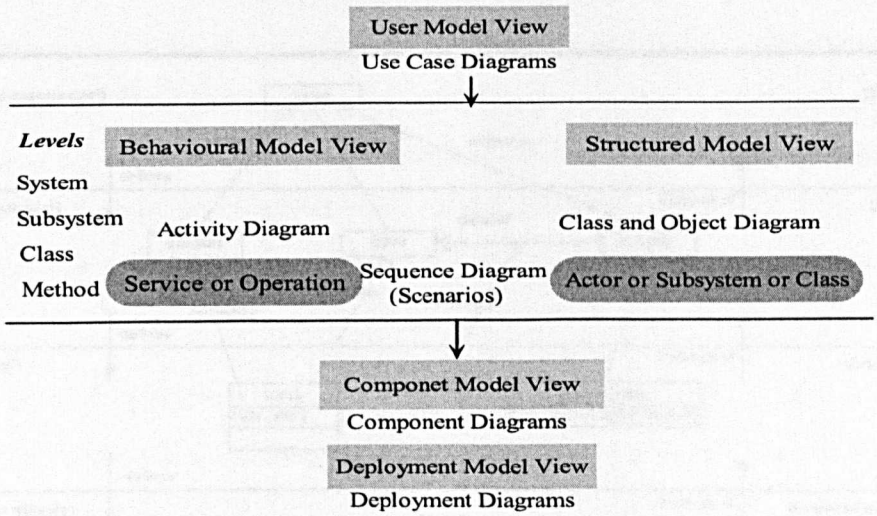


Figure 3-4. UML Model Views at Different Levels of Abstraction [111]

The logical view primarily supports the functional requirements, which models the

translation of the system use cases into functional aspects of the system. The process view is intended to show the concurrent execution aspects of a system and the collaborations needed to support them. It is usually expressed as tasks, threads, or active objects. The implementation view concerns with the representation of the system as modules, libraries, sub-systems and other software components. This implementation view describes the mapping of the design elements into their actual form. The deployment view describes the deployment of the software elements of the system to the hardware elements and the relationships between those hardware elements. The use case view overlaps the other views and plays a special role. It shows the use cases and actors that define the requirements of the system [111].

3.5.2 Meta Object Facility (MOF)

Object Management Group (OMG) presented a four-layered architecture of modelling. Figure 3-5 shows a typical instantiation of the MOF metadata architecture with meta-models for representing UML diagrams and OMG IDL [122]. The Meta-Object Facility (MOF) specification defines an abstract language and a framework for specifying, constructing, and managing technology neutral meta-models. A meta-model is in effect an abstract language for some kind of metadata.

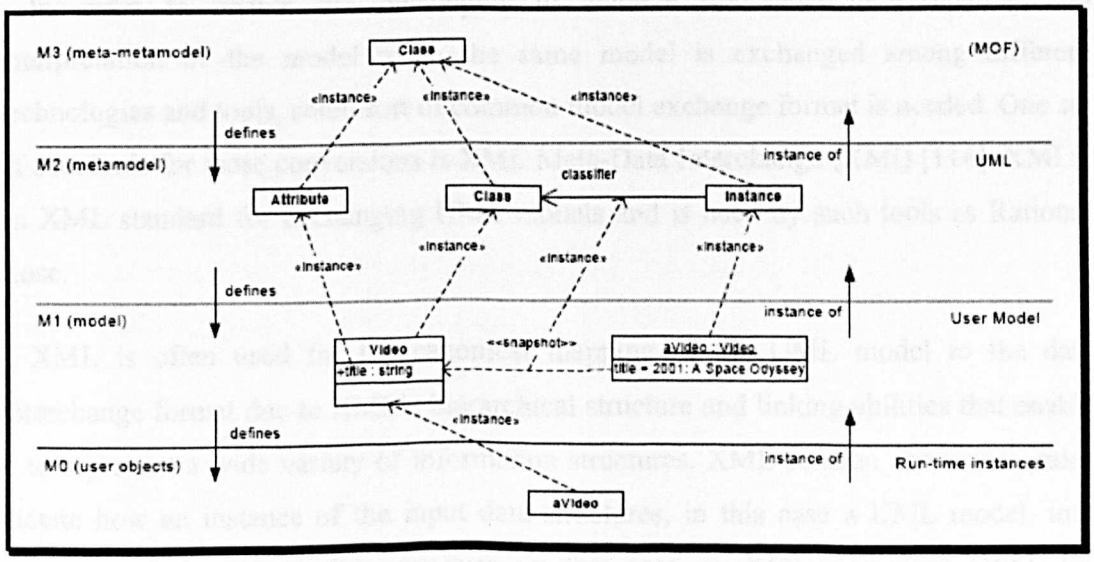


Figure 3-5. Four-layer Metamodel Hierarchy [122]

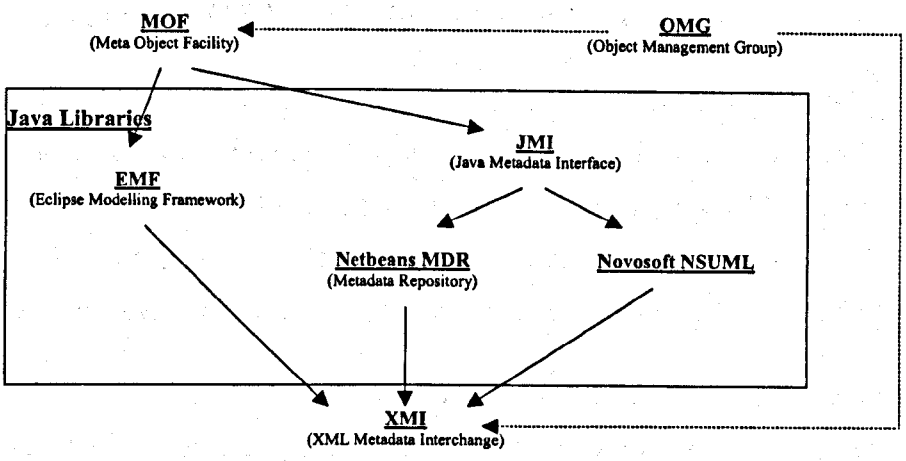


Figure 3-6. Mapping of OMG Standards to Java

In addition, the MOF defines a framework for implementing repositories that hold metadata (e.g., models) described by the meta-models. This framework uses standard technology mappings to transform MOF meta-models into metadata APIs. This gives consistent and interoperable metadata repository APIs for different vendor product and different implementation technologies. Figure 3-6 shows several implementations of the MOF meta-meta-model in Java [17]. All of them have the ability to create a MOF model in the memory and write it to an XML Meta-Data Interchange (XMI) file.

In order to enable the developers to achieve the same understanding and interpretation of the model when the same model is exchanged among different technologies and tools, some sort of common model exchange format is needed. One set of standards for these conversions is XML Meta-Data Interchange (XMI) [114]. XMI is an XML standard for exchanging UML models and is used by such tools as Rational Rose.

XML is often used for the canonical mapping of the UML model to the data interchange format due to XML's hierarchical structure and linking abilities that enable it to represent a wide variety of information structures. XML schema conversion rules dictate how an instance of the input data structures, in this case a UML model, into instances of the output data structure, in this case an XML document [105]. By following these rules, the UML model is incorporated into a XML document.

3.5.3 Executable UML and Model Compilers

The goal of executable UML is to integrate the modelling language and the programming language into one language with a visual as well as a textual syntax and with common semantics. Having such a powerful language would allow the people to execute the design models long before these models had been translated into a programming environment to dramatically eliminate work [102].

Model is at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organisation of the software so that a specification built in Executable UML can be deployed in various software environments without change. MDA depends on the notion of a PIM and PSM. A PIM is independent of its platform(s) and can be built using an executable UML. Executable UML views the PSM as an intermediate graphical form of the code that is dispensable in the case of complete code generation [102].

An executable UML model completely specifies the semantics of a single subject matter, and in that sense, it is indeed a programming language. Yet decisions about the organisation of the hardware and software are abstracted away in an executable UML model, just as decisions about register allocation and stack/heap organisation are abstracted away in the typical compiler. And, just as a typical language compiler makes decisions about register allocation and the like for a specific machine environment, so does an executable UML model compiler turns an executable UML model into an implementation using a set of decisions about the target hardware and software environment. There are many possible executable UML model compilers for different system architectures. Each can compile any executable UML model into an implementation [102].

3.6 Related Projects

3.6.1 Model Integrated Computing (MIC)

Model Integrated Computing (MIC) was developed at the Institute for Software

Integrated Systems, Vanderbilt University since the late 1980s, which is a software and system development approach that advocates the use of domain-specific models to represent relevant aspects of a system. Model Integrated Program Synthesis (MIPS) uses MIC to produce the model, and then from that model produces the computer program that is the executable code (also known as the executable model) of the computer-based system. The advantage of this methodology is that it expedites the design process, supports evolution, eases system maintenance and reduces costs [153].

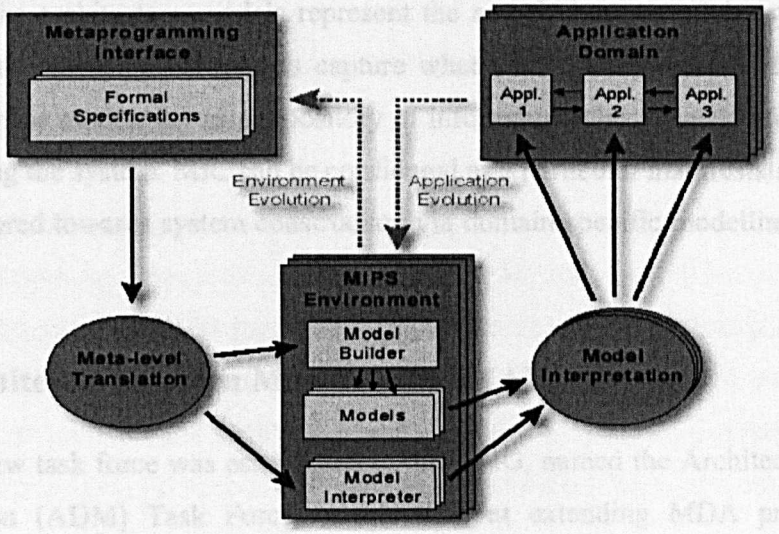


Figure 3-7. MIC Development Cycle [153]

The MIC development cycle (Figure 3-7) starts with the formal specification of a new application domain. The specification proceeds by identifying the domain concepts, their attributes, and relationships among them through a process called meta-modelling. The Generic Modelling Environment (GME) is the main component of the latest generation of MIC technologies. GME provides a framework for creating domain-specific modelling environments [153].

MIC has several layers of specification that are subject to change. Since with MIC the behaviour and structure of the system is specified using models, any changes to that behaviour are made to the models. The right side of Figure 3-7 gives an overview of the MIPS evolution cycles. Application evolution is facilitated by the MIPS environment, which is developed for the domain. Notice that in order to modify the execution models,

changes are made to the domain models. The resulting system may require maintenance/updating not only on the instance level, but also on the meta-model level, known as environment evolution. Environment evolution is required when any formal specification of the domain is changed.

Using MIC technology one can capture the requirements, actual architecture, and the environment of a system in the form of high-level models. The requirement models allow the explicit representation of desired functionalities and/or non-functional properties. The architecture models represent the actual structure of the system to be built, while the environment models capture what the "outside world" of the system looks like. These models act as a repository of information that is needed for analysing and generating the system. MIC can be considered as a particular manifestation of MDA, which is tailored towards system construction via domain-specific modelling languages [153].

3.6.2 Architecture Driven Modernisation (ADM)

In 2003, a new task force was established in the OMG, named the Architecture Driven Modernisation (ADM) Task Force, which aims at extending MDA practices and standards to existing systems [116]. ADM specifically addresses the modernisation of legacy systems in the context of the MDA for the purpose of mining legacy systems, recovering their architecture, identifying inconsistencies and migrating them into new system. The goals of ADM are:

- to revitalisation of existing applications (the ultimate goal),
- to make existing applications more agile,
- to leverage existing OMG modelling standards and the MDA initiative, and
- to consolidate best practices leading to successful modernisation.

MDA uses a top-down approach for developing new systems: from models to systems, while ADM tries to work bottom-up by extracting architectural models from

the existing systems, which can be used in a top-down MDA development process. ADM consists of seven different proposals (RFP's) targeted at seven different activity areas [116]: Knowledge Discovery (KDM); Abstract Syntax Tree (ASTM); Analysis (AP); Metrics (MP); Visualisation (VP); Refactoring (RP); Target Mapping and Transformation (TMT). Currently, only KDM and ASTM have been clearly defined. It is clear that the ADM task force has a long time frame.

3.6.3 Harvesting Project

The work presented in [134] was carried out within a pilot study conducted at a major Dutch insurance company. The objective of this pilot study is to investigate the feasibility of adopting MDA techniques for their legacy systems, in order to safeguard the future maintainability of these systems. The steps have been implemented in a prototype "harvesting" tool that is based on Arcstyler [68]. Arcstyler used the term MDA-Cartridges for the model transformation. Arcstyler has the support of creation and editing of MDA-Cartridges so that one could define or modify his own transformation rules. Arcstyler provides the predefined cartridges for a number of technologies and platforms, such as Java, J2EE and .NET. Steps in "harvesting" approach consist of:

- parsing the source code of the legacy system according to a grammar,
- mapping the abstract syntax trees thus obtained to a grammar model that is defined in the MOF,
- using model to model transformations to turn the grammar model into a generic meta-model, called GenericAST, in which information about software systems can be stored in a language-independent way, and
- mapping the GenericAST models, again using model to model transformations, to UML models that can be either used for code generation or for documentation purposes.

The generic intermediate model used in harvesting project allows people to reuse model to model transformations. For specific harvesters, a transformation can be

developed to generate a generic model. Available transformations and analyses can then be applied to the generic model to get the desired target models. The case study has shown that the prototype implementation of the reverse engineering framework is able to extract models from a production system of 178k Lines Of Code (LOC): UML models have been generated that give insight in structure and behaviour. These models can be used for documentation purposes as well as for (partial) forward engineering.

3.7 Summary

The current researches related to model driven software modernisation have covered a number of areas. In this chapter, the necessary background knowledge for understanding the rest of thesis is introduced.

- Model driven software modernisation should combine both formal and cognitive reverse engineering techniques.
- The WSL has been proved to be a successful approach for program migration.
- Software Architecture Reconstruction can contribute to the architecture recovery by providing a clear goal for reverse engineering. Pattern based and AOP based software evolution are structural methods that utilise the design experience and knowledge.
- Feature models can present the reusable aspects of a domain, which is a missing aspect in other modelling techniques.
- UML is a wide spectrum visual specification language with loosely coupled sublanguages. However, it does not have an integrated model and it is not designed for re-engineering.
- Existing model driven re-engineering projects have taken huge efforts on building transformation rules for automatic model transformations but almost all of them are in initial stage and their successes are limit.

Chapter 4

Proposed Approach

Objectives

- To summarise the rationale for the proposed approach, REMOST (Re-Engineering through MOdel conStruction and Transformation).
- To introduce REMOST architecture.
- To describe REMOST process model.

4.1 Overview

It is certain that the contemporary software development requires software evolution. There is still a gap in its methodological supporting. In this chapter, a unified re-engineering approach, REMOST (Re-Engineering through MOdel conStruction and Transformation), is proposed in line with MDA philosophy. The REMOST approach is a model centric method based on re-engineering techniques. The main goal of the method is to be able to recover consistent and validated models corresponding to the legacy system and transform the recovered models in order to build the modern target system. To achieve this goal, reverse engineering techniques have to be used to understand the legacy system and produce high level system models/views, which will be achieved by analysing, understanding, evaluating and regenerating a legacy system in such a way that the techniques of transformation, decomposition and abstraction are applied.

In following sections, various models, their purpose and their relations are introduced.

The framework of REMOST approach is presented in detail and the architecture and working processes of WML are discussed.

4.2 Framework of REMOST Approach

The foundation of the REMOST approach is based on the construction of a spectrum of WSL-based Modelling Language, known as WML. WML provides reflexive knowledge about the models and expresses multiple layers of abstractions, while allowing integration with legacy infrastructures. A unified framework is built to support the proposed approach as shown in Figure 4-1, which gives the guidance to the re-engineering process. The whole process is divided into separate phases, activities and tasks, and structured into different abstraction levels with different system models.

The left of Figure 4-1 shows the architecture of WML and the possible process when using WML to re-engineer legacy systems. WML introduces four layers to make up a framework: Wide Spectrum Language (WSL) Layer, Common Modelling Language (CML) Layer, Architecture Description Language (ADL) Layer, and Domain Specific Modelling Language (DSML) Layer. The whole process of the proposed approach can roughly be divided into five phases, which are a serial of model constructions and program/model transformations:

- to construct the domain model in DSML,
- to translate from legacy source code to WSL,
- to transform WSL into CML,
- to recover software architecture based on DSML and CML, and
- To map WML into UML.

The right of Figure 4-1 shows the architecture of MDA and the process of applying MDA. The first step is to construct the CIM, which expresses only business functionality and behaviour. Secondly, PIMs are created at the next level include some

aspects of technology even though platform-specific details are absent. Thirdly, the produced PIM is translated to a platform-specific model (PSM), and the last step will generate the application from the PSM using a platform-specific code generator. Although there is no restriction to the languages used to represent models of MDA, UML is the favourite choice by far.

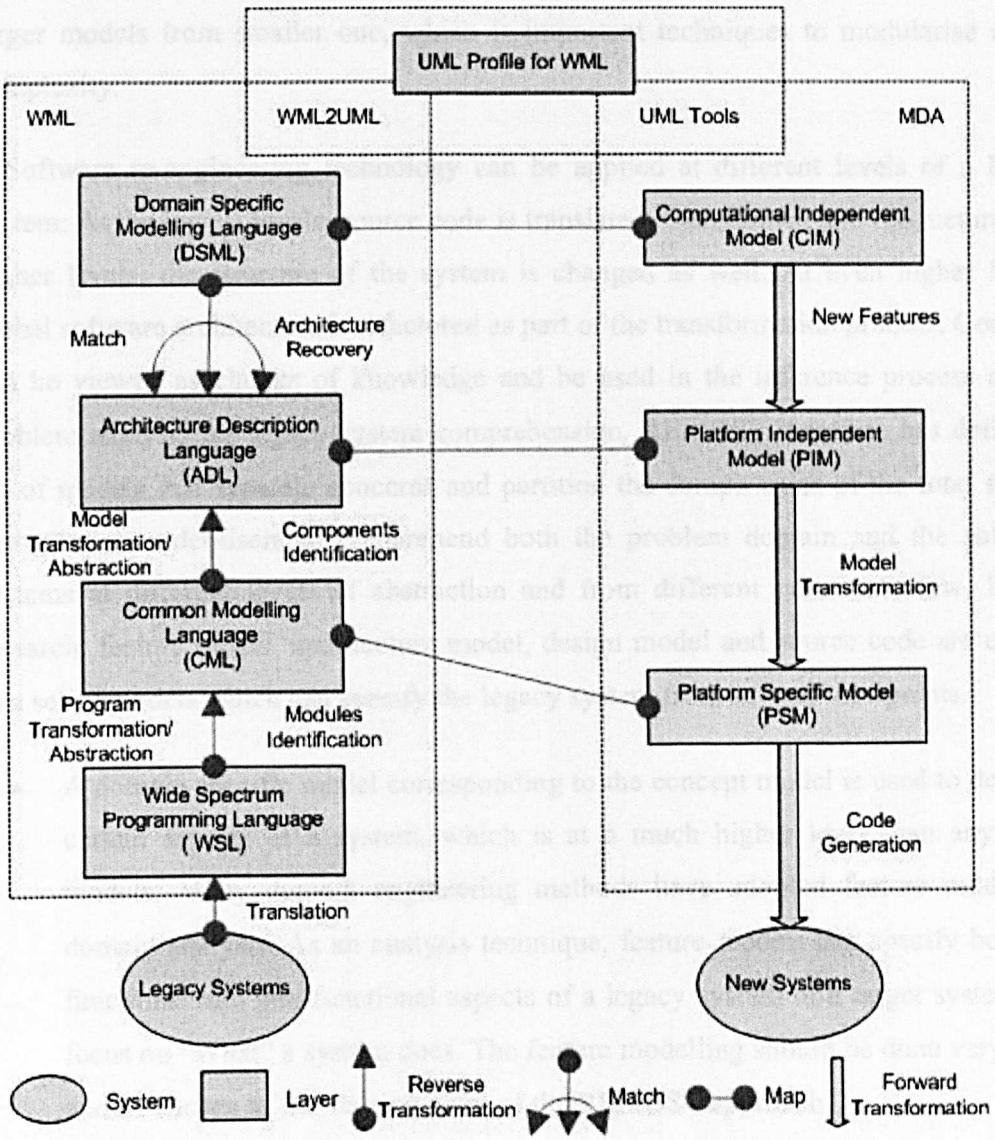


Figure 4-1. Framework of REMOST Approach

4.2.1 Choice of Models

Modern software systems and the problems they solve are far too complex for any human being to understand as a whole. To support the analysis of complex systems, a system description is made of numerous models. Each model represents a different level of abstraction. The hierarchical structure provides ways to organise the models, to build larger models from smaller one, which is important techniques to modularise model complexity.

Software re-engineering technology can be applied at different levels of a legacy system. At the lowest levels, source code is translated, transformed and restructured. At higher levels, the structure of the system is changed as well. At even higher levels, global software architecture is refactored as part of the transformation process. Concepts can be viewed as chunks of knowledge and be used in the inference process during problem analysis for legacy system comprehension. REMOST therefore has defined a set of models that separate concerns and partition the complexities of the total system that allows modernisers to comprehend both the problem domain and the software systems at different levels of abstraction and from different points of view. In this research, feature model, architecture model, design model and source code are chosen as a set of models which can specify the legacy system from several viewpoints.

- A domain specific model corresponding to the concept model is used to describe certain aspects of a system, which is at a much higher level than any other models. Many domain engineering methods have adopted feature models in domain analysis. As an analysis technique, feature models can specify both the functional and non-functional aspects of a legacy system or a target system and focus on "What" a system does. The feature modelling should be done very early and be chosen as the starting point of the REMOST approach.

- Software architecture is also a high-level abstraction of the software system but focuses on "How" a system organises its structure and implements its behaviour. The software architecture models specify the components and their collaboration in terms of layers and subsystems. Aspects and architecture patterns can aid to

understand the software architecture in a structural view.

- A design model is chosen to illustrate the detail design of a legacy system or a target system. The design pattern usage is encouraged because design patterns provide the flexibility and extensibility while helps framework developers understand the framework by serving as a common vocabulary between the framework builders and the application developers [47].

- The source code is also a kind of model, describing the implementation details on a system. It is, of course, a highly platform-specific model, but a model nevertheless.

4.2.2 Architecture of WML

In order to support the above models, WSL is further extended as WML (WSL-based Modelling Language), which is designed to describe unambiguous specification/descriptive model of software systems. Since WML is an extension of WSL, it integrates the modelling language and the programming language into one language. In this way, the same modularisation constructs can be used to structure models and programs with a gradual transition and the consistent of modelling language and programming language make it possible to trace the changes between source code and models.

4.2.2.1 Common Modelling Language

Common Modelling Language (CML) in WML is designed to describe specification and/or descriptive model of software systems. The CML is a textual modelling language, which supports conventional features of most modelling languages. A small set of linguistic primitives and objects are identified as both required and sufficient building blocks of the meta-language, which offers adequate modelling concepts as a formal foundation for the CML. These concepts includes class, object, behaviour, message, process, etc., from which models of a system are composed.

4.2.2.2 Architecture Description Language

Architecture Description Language (ADL) is designed to describe platform-independent model of software systems, which provides a formal description of software architectures. ADL in WML consists of three elements: component, connector and configuration. It focuses on the structure and interrelationships between system components and provides higher level views than components, which aims at designing systems with coarser-grained elements and their overall interconnection structure. The formal approach of ADLs allows the application of analysis, which plays a significant role during software evolution. It is not only the starting point for system design and development, but the desired outcome of reverse engineering.

4.2.2.3 Domain Specific Modelling Language

In order to apply a domain model in re-engineering, design of Domain-Specific Modelling Language (DSML) is required, which is highly domain-specific, very high level, and formally specified. DSML captures essential domain knowledge and distils the experience and knowledge that is implicit in a number of legacy systems into concise DSML-specified models. Domain models can then be processed by tools and used to (re)generate software systems. In this research, Feature Description Language (FDL) is used to specify a feature model as a domain model.

4.2.3 *Meta*WML for Program and Model Transformation

*Meta*WML is the extension of *Meta*WSL and focuses on the model transformation. Besides having all the standard features of programming and modelling language, *Meta*WML contains commands, functions and routines for operating on program and model elements, which gives the user a tool for analysing, rewriting, and simplifying programs/models. So *Meta*WML supports both program and model transformation.

The meta-model of a language, also known as the abstract syntax, is a description of all the concepts that can be used in that language. A *Meta*WML transformation program is operated on meta-model level and composed of rules that define how elements of the

source system are matched and navigated to create and initialise the elements of the target system. Besides basic program transformations, *Meta*WML defines an additional model querying facilities, action primitives and metrics functions that enable to specify model transformation easily.

4.2.4 Mixed Approach for Round-trip Re-engineering

In this research, a round-trip re-engineering approach is proposed in line with MDA. "Round-trip" means that to produce a model and generate code from it, and then whenever the code is changed, a new model will be synchronised. It is therefore supporting forwards and backwards generation. There are four different methods to understanding and maintaining a large legacy system, namely "top-down", "bottom-up", "outside-in" and "middle-out" methods [164].

A "top-down" method is a way of comprehending the large programs by providing a hierarchical structure. A "bottom-up" method starts by analysing the lowest level routines, which are used to understand higher level routines, abstract data types and so on. The higher level routines and abstract data types will be increasingly domain specific and problem specific. An "outside-in" method is actually combination of top down and bottom up methods working in parallel. A "middle out" method means to start in the middle of the abstraction hierarchy and work outwards to both higher and lower levels of abstraction. This method is a kind of heuristic method, involving a repetitive process of making, testing and adjusting hypotheses until the entire system can be explained consistently.

The "top-down" method requires that the re-engineer has an almost complete concept of the system while the "bottom-up" method requires that someone knows how to fit the lower-level routines into the big picture. The "outside-in" method combines both the advantage and disadvantage of "top-down" and "bottom-up" methods. It is hoped that the two methods will meet in the middle. The main merit of "middle out" method is that the re-engineered process can start at any level according to the different application context while the concurrent engineering is also possible [164].

REMOST methodology adopts a mixed approach, where WML is introduced to define the "middle layers" to meet the requirements of applying "middle out" methods. It should be noticed that the "top-down", "bottom-up" and "outside-in" methods can still be used in the whole re-engineering process and at each middle layer as well and WML is designed to make it as easy as possible for developers to have round-trip engineering.

4.2.5 Software Modernisation Process

The REMOST models described in previous sections are the deliverables produced by the activities performed during the software re-engineering processes. These models identify and set focus for the activities and thereby drive the whole modernisation process. In principle, all the models can be considered during every phase and iteration. However, since each company and each project is unique, these models must also be able to tailor to their specific needs.

4.2.5.1 Domain Model Construction

The starting point for the REMOST approach is construction of a conceptual model. The concept model is written in DSML that is tailored towards the specific domain. DSML specifies domain-specific terms and their relations, defines the boundaries of the domain. Feature modelling techniques are adopted in this research which takes into account both functional and non-functional properties.

4.2.5.2 Source Code to WSL Translation

To translate source code into WSL code has been researched and discussed clearly in [180]. Once the source program has been captured in WSL, there are a large number of restructuring and simplifying program transformations that can be applied automatically to clean up the code, unscramble the structure, and delete redundant code. The result is a structured program consisting of a hierarchy of single-entry, single-exit procedures.

4.2.5.3 WSL to CML Translation

To translate WSL into CML is a transformation from grammarware to modelware [173]. The transformation rules are applied and an initial model is generated in CML. The proposed method in this research focuses on how to analyse WSL code for abstraction, transformation and representation with models.

4.2.5.4 Architecture Recovery

At heart of the REMOST methodology lies the architecture recovery phase. This phase compares the results of the "top-down" and "bottom-up" phase, and performs an incremental recovery of the system's architecture through establishment and verification of hypotheses.

Top-down Phase:

The "top-down" method resembles the "classical" MDA process, starting from a CIM and transforming it into a PIM. A hypothesis describes a supposed relationship among a feature and an architectural element. Feature modelling structures the system's functionality by detecting the relations between source code elements and requirements. Feature models fill the gap between requirements and the solution, which provide an extra model between requirements specifications and design models. By linking features to requirements, detailed information from the problem domain is reachable. These links are built using traceability links. Tracing these relations may lead to the recovery of various architectural elements. In this way, the system's feature model supports program comprehension and architectural recovery and the results of the analysis refine a hypothesis through architectural description.

Bottom-up Phase:

The "bottom-up" method reverses "top-down" process, extracting information from legacy code and data and then abstracting them into an architecture model in ADL to build initial legacy system architecture that is a legacy PIM. Reverse engineering techniques here are used to produce abstracted views/models. The most sophisticated

technical approach to this step is to decompose the legacy system into components in such a way that these components become reusable across different applications. There are two important kinds of decomposition concept: modular decomposition and aspectual decomposition.

- Modular decomposition involves decomposing systems into hierarchical units such as modules, components, objects, functions, procedures, and so on. The "hierarchical" indicates that a unit may contain other units recursively. The goal is to achieve high cohesion within the units and minimal coupling between the units.

- The main idea behind aspectual decomposition is to organise the description of a concept into a set of perspectives, where each perspective concerns itself with a different aspect and none of which is itself sufficient to describe the entire concept. An important property of such decomposition is that each perspective yields a model with a different structure, and all of these models refer to the same concept as crosscutting.

Given the subdivision of the legacy system resulting from decomposition, a class of transformation rules have been dedicated to group relevant components to form clusters for further restructuring.

Matching Phase:

During the architecture recovery phase, both the new PIM and legacy PIM are matched. During matching, a source specification is compared to a target specification resulting in a collection of mappings between elements of both specifications.

4.2.5.5 A Bridge between WML and UML

In order to integrate with MDA environment, WML should be mapped onto semi-formal UML diagrams which can be presented visually and translated into XMI for information exchange. WML provides a development framework that supports rigorous model analysis when it comes to generating models/implementations through

transformations, while UML preserves useful features of the graphical modelling techniques.

4.3 Summary

In this chapter, a unified re-engineering approach, REMOST (**Re**-Engineering through **MO**del con**S**truction and **T**ransformation), is proposed for software modernisation.

- The REMOST methodology is inline with MDA philosophy. In particular, the "top down" method starts from a domain model and refines it into a domain target system architecture model; the "bottom up" method extracts design models from the code and abstracts them into a legacy system architecture model. Both the legacy and the new architecture models are compared and mapped to each other, resulting in recovered architecture.

- The REMOST approach supports round trip re-engineering with a mixed approach, including "top-down", "bottom-up", "outside-in" and "middle-out" methods. The "middle-out" approach starts in the middle of the abstraction hierarchy and works outwards to both higher and lower levels of abstraction.

- The multiplicity of the abstraction levels is appropriate to the modelling approach. WML is defined with a different viewpoint for each abstraction level.

- Model transformations are of critical importance during several stages in the methodology. *Meta*WML is a model transformation language which can be used to specify transformation rules and facilitate the model transformation.

- WML can be unified with UML so that the system models can be presented visually and supported by UML-compatible tools.

Chapter 5

WSL-based Modelling Language

Objectives

- To analyse the design goals and discuss characteristics of WML.
- To define the syntax of WML.
- To illustrate some examples of WML.

This chapter provides a reference of WSL-based Modelling Language (WML), including Common Modelling Language (CML), Architecture Description Language (ADL) and Feature Description Language (FDL). Since WML is an extension of Wide Spectrum Language (WSL), it embodies all experience and previous work in the syntax, semantics and tool that support the new modelling language. First, the design goals of WML are analysed and characteristics of WML are discussed. Then, the syntax of WML is formally defined with Extended Backus-Naur Form (EBNF) and car examples are illustrated to describe the usage of WML. A summary of the WML grammar appears in Appendix A.

5.1 Design Goals and Characteristics of WML

The design goals of WML embody the goals for designing modelling language [123] and the goals for supporting software re-engineering effectively:

- Simplicity: No unnecessary complexity is included in the language. It should be the leading goals in designing a modelling language. The modelling language

should decrease unnecessary notions and use simple syntax and semantics. This leads to easiness of study and use.

- Modularity: The modularity is a fundamental characteristic to ensure well-structured models.

- Flexibility: The language can be applied to a variety of software systems and should not be confined to a kind of specific software.

- Scalability: The language can model both large and small systems.

- Expressiveness: The language can accurately reflect the models.

- Seamlessness: Seamlessness allows the mapping of abstraction in the problem space to implementation in the solution space without changing notation. The same abstraction can be used throughout development.

- Reversibility: Implementation changes can be propagated into the model. The goal of reversibility contributes to the production of maintainable software, and to producing better documentation for software systems.

- Supportability: Supportability states that a modelling language should be designed to be implementable and supportable by software tools. The tools should provide support for developing software and provide support for ensuring that the models being produced are consistent.

It is difficult to satisfy each goal in isolation. The design of WML focuses on the purpose of reverse engineering that WML must have certain characteristics:

- WML should provide a concise mechanism for describing the fundamental abstraction so that it can be applicable to handle complexity systems.

  ✓ WML should be extended from WSL that both programming languages and modelling languages can be used for describing software systems at different levels of abstraction. WML that is unified with WSL makes it easy to bridge

the gap between programming languages and modelling languages.

- ✓ WML should describe all of the elements (e.g. classes, objects, processes, etc.) and their relationships (e.g. generalisation, aggregation, etc.) at appropriate levels of detail and in an appropriate form.
- ✓ WML should provide a grouping mechanism to hide their details. More concepts such as encapsulation and nesting are needed to manage complexity.
- ✓ WML should provide views/aspects mechanism to collect the related concerns, which is another kind of abstraction dimension.

- WML should support self-documenting systems. Reversibility, combined with seamlessness, allows programs and models to be kept in synchronisation, and thus helps create and maintain system documentation. Changes made to models can be reflected in code and changes made to code can be reflected in changed models automatically.
- Finally, WML should be mapped to visual descriptions easily: for large systems, diagrammatic representation can help in understanding the source code. So it is expected that the definition of WML should be as close as possible to the definition of UML that WML can be mapped to UML easily.

WML is designed in four layers to make up a framework: Wide Spectrum Language (WSL), Common Modelling Language (CML), Architecture Description Language (ADL), and Feature Description Language (FDL) as Domain Specific Modelling Language (DSML). Developing a new language from scratch is discarded, so WML is designed based on some existing languages. The key point is how to define these languages so that all of them are linked together and used in a consistent way. In following sections, CML, ADL and FDL are presented in detail.

5.2 WML Common Modelling Language

This section discusses fundamental concepts and syntax of WML Common Modelling Language (CML). UOL [120] is chosen as the basis to develop CML. Concepts in OCL [117], POOSL [51], SDL [70] and WSL [87, 165, 180] are also used as an extention. The CML is a textual modelling language, which offers adequate modelling concepts as a foundation for the WML. These concepts are the basic building blocks including class, object, behaviour, message, process, etc., from which models of a system are composed.

5.2.1 CML Syntax

The following summarises basic CML syntax. For the sake of simplicity, some unimportant syntax components are omitted here. Please see the Appendix A.4 for language details.

5.2.1.1 Syntax for Class and Process Definition

Most of CML constructs are equivalent to UML entities. The process is also declared as class and hence a process class can inherit characteristics from other process class. CML describes all of the elements (e.g. classes, objects, processes, etc.) at appropriate levels of detail and in an appropriate form.

```
                Features ::= ( <FEATURE> "{" Visibility "}" Feature_list <END> )*
              Visibility ::= ( <ANY> | <NONE> | Classifier_list )
         Classifier_list ::= Classifier_name ( "," Classifier_name )*
         Classifier_name ::= <IDENTIFIER>
            Feature_list ::= Feature_declaration ( ";" Feature_declaration )*
   Feature_declaration ::= ( Attribute | Operation_declaration | Method_declaration )
               Attribute ::= <IDENTIFIER> Type_mark Extension_use ( <IS>
                             Initial_value )?
           Initial_value ::= Expression
Operation_declaration ::= New_operation Operation_body
          New_operation ::= <OPERATION> <IDENTIFIER>
          Operation_body ::= ( Formal_arguments )? ( Type_mark )? Extension_use ( <IS>
                             Specification )?
       Formal_arguments ::= "(" Entity_declaration_list ")"
Entity_declaration_list ::= Entity_declaration_group ( ";" Entity_declaration_group )*
```

```
Entity_declaration_group ::= ( Parameter_kind )? <IDENTIFIER> ( "," <IDENTIFIER> )* Type_mark
Parameter_kind ::= ( <IN> | <OUT> | <INOUT> )
Specification ::= <TEXT_MULTILINE>
Method_declaration ::= New_method Method_body
New_method ::= <IDENTIFIER>
Method_body ::= ( Formal_arguments )? ( Type_mark )? Extension_use ( Specification_use )? ( <IS> Routine )?
Specification_use ::= ( <IDENTIFIER> | <TEXT_MULTILINE> )
Routine ::= <TEXT_MULTILINE>
Interface_declaration ::= Interface_header ( Formal_generics )? ( Viewed_with )? Extension_use ( Inheritance )? ( Interface_operation_declaration )* <END>
Interface_header ::= <INTERFACE> <IDENTIFIER>
Interface_operation_declaration ::= <FEATURE> "{" Visibility "}" ( Operation_declaration )*
Class_declaration ::= Class_header ( Formal_generics )? ( Viewed_with )? Extension_use ( Inheritance )? ( State_machine )? Features ( Use_of_constraint )? <END>
Class_header ::= ( <DEFERRED> )? <CLASS> Class_name
Class_name ::= <IDENTIFIER>
Process_name ::= <IDENTIFIER>
Init_Parameter ::= Declarations
Declarations ::= Declaration ( "," Declaration )*
Declaration ::= <IDENTIFIER> ":" <IDENTIFIER>
Port_name ::= <IDENTIFIER>
Message_Sig ::= Message_name ":" Port_name ":" Process_name
Message_name ::= <IDENTIFIER>
Process_Method ::= Method_name "(" Declarations ")"
Thread_declaration ::= <Thread> Thread_name "In" Process_name ( "(" ( Init_Parameter )? ")" )*
Process_declaration ::= <ProcessClass> Process_name ( "(" ( Init_Parameter )? ")" )* ( "Extends" Process_name )? ( <Port> ( Port_name )? )* ( <message> ( Message_Sig )? )* ( <method> ( Process_Method )? )* <END>
```

List 5-1. Syntax for Class and Process Definition

5.2.1.2 Syntax for Relationship

CML relationships can express almost all the UML (association, aggregation, dependency and generalisation) relations at appropriate levels of detail and in an appropriate form.

```
Inheritance ::= <INHERIT> Parent ( ";" Parent )*
Parent ::= Class_type ( Feature_adaptation )?
Class_type ::= Class_name Actual_generics
Actual_generics ::= Type ( "," Type )* "]"
Type ::= ( Class_type | Class_type_expanded | Anchored | Bit_type )
Relation_declaration ::= ( <DEFERRED> )? <RELATION> Relation_name
    ( Formal_generics )? ( Relation_inheritance )? ( Link_list )?
    ( Delegation_list )? Features <END>
Relation_name ::= Element_name
Relation_inheritance ::= <INHERIT> Parent_relation ( ";" Parent_relation )*
Parent_relation ::= Relation_type ( <ADAPTATION>
    Relation_feature_adaptation )?
Relation_type ::= Relation_path ( Actual_generics )?
Relation_path ::= Element_path
Relation_feature_adaptation ::= ( Rename )? ( New_exports )? ( Undefine )? ( Relation_redefine )?
    ( Select )? <END>
Relation_redefine ::= <REDEFINE> ( Feature_list | Redefine_with_list )
Redefine_with_list ::= Redefine_pair ( "," Redefine_pair )*
Redefine_pair ::= Feature_name <WITH> Feature_name
Link_list ::= <LINK> ( Type_link_two_list | Dependency ( ","
    Dependency )* )
Type_link_two_list ::= Type_link ( "," Type_link )+
Type_link ::= ( Classifier_name | <THIS> )
Dependency ::= Element_path <TO> Element_path
Delegation_list ::= Delegation ( Deleg_relation_redefine )? ( "," Delegation
    ( Deleg_relation_redefine )? )*
Deleg_relation_redefine ::= Relation_redefine <END>
Delegation ::= ( Type_link )? "." Feature_name <LIKE> ( Type_link )? "."
    Feature_name
```

List 5-2. Syntax for Relationship

5.2.1.3 Syntax for Model Management

The model management provides abstraction of CML. CML provides a grouping mechanism to hide their details and also provides views/aspects mechanism to collect the related concerns, which is another kind of abstraction dimension.

```
CML ::= <MODEL> Model_name Extension_use ( <DIAGRAMS>
    View_element_decl_list )? ( Package_declaration |
    Subsystem_declaration | Actor_declaration )* <END> <EOF>
```

```
                Model_name ::= <IDENTIFIER>
  View_element_decl_list ::= View_element_declaration ( "," View_element_declaration )*
View_element_declaration ::= View_element_name ":" View_element_kind
      View_element_name ::= <IDENTIFIER>
       View_element_kind ::= <IDENTIFIER>
     Package_declaration ::= <PACKAGE> Package_name Extension_use ( Viewed_with )?
                             ( Package_inheritance )? ( Package_import )?
                             ( Package_element_decl_list )? <END>
            Package_name ::= <IDENTIFIER>
             Viewed_with ::= <VIEWED> <WITH> View_element_name_list
  View_element_name_list ::= View_element_name ( Position )? ( "," View_element_name
                             ( Position )? )*
                Position ::= "(" <INTEGER_LITERAL> "," <INTEGER_LITERAL> ( ","
                             <INTEGER_LITERAL> )? ")"
     Package_inheritance ::= <INHERIT> Package_name ( "," Package_name )*
          Package_import ::= <IMPORT> Package_import_elem ( "," Package_import_elem )*
    Package_import_elem ::= Visibility Element_path ( <AS> Alias )? <FROM> Package_name
            Element_path ::= Element_name ( "::" Element_name )*
            Element_name ::= <IDENTIFIER>
                   Alias ::= Element_name
Package_element_decl_list ::= <IS> ( Package_element_decl )+
    Package_element_decl ::= ( Actor_declaration | CML | Package_declaration |
                             Interface_declaration | Class_declaration | Relation_declaration |
                             Stereotype_declaration | Constraint_declaration | Tagged_values |
                             Usecase_abstraction | Activity_model | Collaboration_declaration |
                             Comment_definition | Actor_or_exception Light_body |
                             Component_or_node Ultra_ligth_body )
  Comment_definition ::= <TEXT_MULTILINE> <ATTACHED> <TO> Element_name
```

List 5-3. Syntax for Model Management

5.2.2 Sample of CML

In the following example, a car production package is selected to present how to express model of car type, owner, and manufacturer in CML (List 5-4).

```
package car_production
  viewed with carClassDiagram
  public
    class CarType
    steretyped with type
    creation Make
    features { NONE }
      plate : string
      model : string
```

```
        trademark : string
        maxSpeed : integer
        numberSeat : integer
      features { ANY }
        Make ( plate : string , modelVal : string , trademark: string)
        GetModel ( ):string
        SetModel ( modelVal : string )
      end -- CarType
      class Car
        inherit carType
        features
          creation make ( plate : string , modelVal : string , trademark: string) is
            do ...
          end
          ...
          constrained by
            text "plate /= void"
      end Car
      class Person
        ...
      end -- Person
      relation is_owner
        ...
      end is_owner
      relation is_manufacturer
        ...
      end is_manufacturer
  end
end
```

List 5-4. An Example of Car Production Package in CML

5.3 WML Architecture Description Language

This section discusses fundamental concepts of WML Architecture Description Language (ADL). The ADL defined in this research is based on Acme ADL [1, 48], which can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools. The Acme ADL is chosen as basis of WML ADL for three reasons [1]:

- **Architectural interchange.** By providing a generic interchange format for architectural designs, Acme allows architectural tool developers to readily integrate their tools with other complementary tools. Likewise, architects using Acme-compliant tools have a broader array of analysis and design tools available

at their disposal than architects locked into a single ADL.

- **Extensible foundation for new architecture design and analysis tools.** Acme can mitigate the cost and difficulty of building architectural tools by providing a language and toolkit to use as a foundation for building tools. Further, Acme ADL as a generic interchange language allows tools developed using Acme as their native architectural representation to be compatible with a broad variety of existing architecture description languages and toolsets with little or no additional developer effort.

- **Architecture Description.** Acme ADL provides a straightforward set of language constructs for describing architectural structure, architectural types and styles, and annotated properties of the architectural elements.

Acme ADL defines 7 basic element types: components, connectors, systems, ports, roles, representations, and representation maps. Figure 5-1 shows a description of the five most important elements and their relationships.

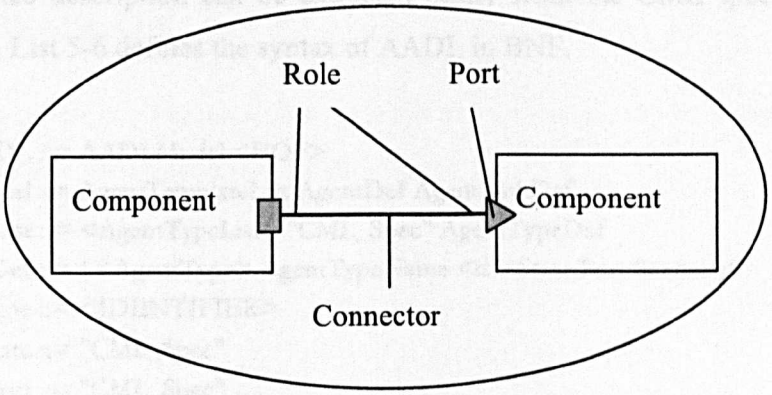


Figure 5-1. Elements of an ADL Description

List 5-5 shows a small architecture represented with ADL, describing the architecture of after service system for car production.

```
System car_cs = {
  Component customer = { Port sendRequest }
  Component salesman = { Port receiveRequest }
  Connector rpc = { Roles {caller, callee} }
  Attachments : {
```

```
        client.sendRequest to rpc.caller ;
        server.receiveRequest to rpc.callee
    }
}
```

List 5-5. Sample Code of ADL

5.3.1 ADL Syntax Extension

WML ADL syntax is the same as Acme ADL. Please refer to the Appendix A.4 for language details. In this Section, an extension of ADL language, Agent-oriented Architecture Description Language (AADL), is designed and illustrated, showing that a design of ADL can be extended for presenting special architecture.

AADL is structured like general ADL, covering agent, connector and configuration, which is suitable for agent-based software modernisation. AADL expresses the configuration of interconnected agents and makes reference to services offered at agent interfaces, while referring to the actual internal composition of single agent. The key ideas and core techniques are that CML is integrated into AADL so that the agent's state and service description can be extracted easily from the CML specification of legacy system. List 5-6 defines the syntax of AADL in BNF.

```
          AADL ::= AADLModel <EOF>
     AADLModel ::= AgentTemplateList AgentDef AgentLinkDef
AgentTemplateList ::= <AgentTypeList> "CML_Spec" AgentTypeDef
    AgentTypeDef ::= ( <AgentType> AgentTypeName <is> State Port Service )*
  AgentTypeName ::= <IDENTIFIER>
          State ::= "CML_Spec"
           Port ::= "CML_Spec"
        Service ::= <Requires> ( ServiceList )? ";" <Provides> ( ServiceList )? ";" <Tasks>
                    ( TaskList )? ";"
    ServiceList ::= "CML_Spec"
       TaskList ::= "CML_Spec"
       AgentDef ::= <Agent_instances> <are> ":"
                    ( AgentName <Instantiates> AgentLinkDef ";" )*
      AgentName ::= <IDENTIFIER>
  AgentLinkDef ::= Link ( "," Link )*
           Link ::= One_OneLink
                  | One_ManyLink
```

```
                | Many_OneLink
   One_OneLink ::= ProvideService "->" RequestService
  One_ManyLink ::= ProvideService "->" RequestServiceList
  Many_OneLink ::= ProvideServiceList "->" RequestService
ProvideServiceList ::= ( ProvideService )*
RequestServiceList ::= ( RequestService )*
   ProvideService ::= <IDENTIFIER>
   RequestService ::= <IDENTIFIER>
```

List 5-6. Syntax of Agent-oriented Architecture Description Language

The above descriptions make agents as flexible as possible and avoid committing to a specific agent type. Services define functionalities that the agent provides or requires for accomplishing its purpose. Some services can be exported to external users or other MAS. On the other hand, some services can be imported from outside the MAS. Services typically depend on the domain of application. Each link connects a service provided by an agent to a service required by another agent. <CML_Spec> means that the value can be obtained from the CML specification.

5.3.2 Sample of AADL

In this subsection, car production system is selected as a sample to show how to express an agent-based architecture in AADL (List 5-7).

```
Architecture Car_Production is
{
  AgentTypeList
    Dispatcher, Worker
  AgentType Dispatcher is
    State
      SendQueue, ReceiveQueue
    Port
      SendPort, ReceivePort
    Services
      requires;
      provides AssignTasks;
      Tasks
        Setup();
        AssignTasks();
        tideUp();
      ...
  Agent_instances are
```

```
        D1 instantiates Dispatcher;
        worker1 instantiates Worker;
        ...;
        workerm instantiates Worker;
        ...
}
```

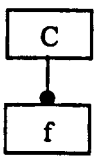
List 5-7. Sample Code of AADL

5.4 WML Domain Specific Modelling Language

As discussed in previous chapters, domain model is mainly defined by feature model. A key element of the feature model is the feature diagram, which is a graphical notation for describing dependencies between features. However, the lack of precision and the ambiguous descriptions of feature diagram have prevented them from wide adoption. To get a better understanding of feature diagrams and enable creation of automatic tools for processing feature diagrams, a textual representation is preferable. In this section, Feature Description Language (FDL) [31] is adopted as part of WML.

5.4.1 Feature Diagrams

A feature model consists of a feature diagram and other associated information (such as rationale, constraints and dependency rules). A feature diagram provides a graphical tree-like notation that shows the hierarchical organisation of features. The root of the tree represents a concept node. All other nodes represent different types of features. Table 5-1 provides an overview of graphical notation introduced in [29]. Assuming that the concept C is selected, feature relationships are defined as follows:

Type	Graphic Notation	Textual Notation
Mandatory		C: All(f)

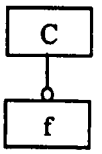
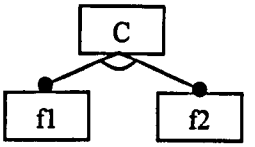
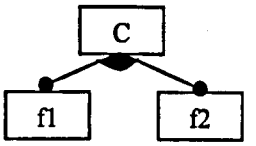
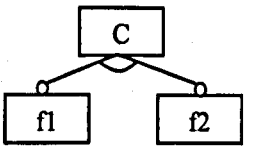
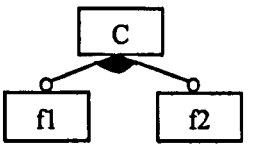
Optional		C: f?
Alternative		C: Oneof(f1, f2)
Or		C: Moreof(f1, f2)
Optional Alternative		C: Oneof(f1?, f2?)
Optional Or		C: Moreof(f1?, f2?)

Table 5-1. Textual Notations for Feature Diagram

- Mandatory - The feature must be included into the description of a concept instance.
- Optional - The feature may or may not be included into the description of a concept instance.
- Alternative-Exactly one feature from a set of features can be included into the description of a concept instance.
- Or - One or more features from a set of features can be included into the description of a concept instance.

- Optional Alternative - At most one feature from a set of features can be included into the description of a concept instance.

- Optional Or - One or more feature from a set of features may or may not be included into the description of a concept instance.

5.4.2 Feature Normalisation

There are two other types of features in a feature diagram, i.e., optional-alternative and optional-or features. An optional-alternative feature type denotes that one or more features in a set of alternative features are optional. From a concept instance point of view, it has the same result as all the features in the alternative set are optional. An optional-or feature type denotes that one or more features in a set of or-features is optional. This is the same as all the feature in the or-feature set are optional, which can be further simplified as each feature in the or-feature set is optional individually. Thus the optional-or features is a redundant feature type, which can be replaced by a set of individual optional features. The above is called normalisation [29] on feature diagrams. Therefore, given any feature model, it can be represented by its normalised form that only contains five possible different type of features, i.e., Mandatory, Optional, Alternative, Or and Optional Alternative.

5.4.3 Textual Notation for Feature Diagrams

There is an increasing need for methods and tools that can support feature model analysis. Feature model may evolve when the knowledge of the domain increases. Thus when features are changed, such tools are required to check if a feature configuration is still valid. As the number of feature increases, an automated method and tool is needed. To enable the creation of automatic tools for processing feature diagrams, a textual notation is preferable. The feature diagram is formally described in BNF syntax in List 5-8.

Lexical syntax:
CompositeFeatureName ::= [A-Z]{[a-zA-Z0-9]}
AtomicFeatureName ::= [a-z]{[a-zA-Z0-9]}

```
Context-free syntax:
FeatureModel ::= {FeatureDefinition}{Constraint}
FeatureDefinition ::= FeatureName ":" FeatureExpression
FeatureName ::= CompositeFeatureName
                |AtomicFeatureName
FeatureExpression ::= <All> (FeatureList)
                | <Oneof> (FeatureList)
                | <Moreof> (FeatureList)
                | (FeatureExpression) "?"
                | FeatureName
FeatureList ::= {FeatureExpression} ( "," FeatureExpression)*
Constraint ::= DiagramConstraint
                | UserConstraint
DiagramConstraint ::= AtomicFeatureName <Requires> AtomicFeatureName
                | AtomicFeatureName <Excludes> AtomicFeatureName
UserConstraint ::= <Include> AtomicFeatureName
                |<Exclude> AtomicFeatureName
```

List 5-8. BNF Rules for FDL

The FDL provides a formal basis for the structural analysis of feature models. The feature hierarchy is established similar to a well structured functional decomposition of the system's architecture. Feature is a dimension of concern that represents distinguishable characteristics of a concept. A concept consists of a set of related features with constraints.

The features which cannot be further subdivided in other features are called atomic features. The features that are defined in terms of other features are called composite features. It is the convention that names of atomic features start with a lower case letter and names of composite features start with an upper case letter. An FDL definition consists of a number of feature definitions: a feature name followed by ":" and a feature expression. A feature expression can consist of:

- an atomic feature,
- a composite feature: a named feature whose definition appears elsewhere,
- an optional feature: a feature expression followed by "?",
- mandatory features: a list of feature expressions enclosed in All( ),

- alternative features: a list of feature expressions enclosed in Oneof( ),
- non-exclusive selection of features: a list of feature expressions enclosed in Moreof( ),

A feature model not only consists of the relationships presented in a feature diagram, but also includes additional constraints among the features that indicate valid combinations in a feature model. Some features are dependent on the presence of other features. A constraint can have one of the following forms:

- A1 Requires A2: if feature A1 is present, then feature A2 should be present as well,
- A1 Excludes A2: if feature A1 is present, then feature A2 should not be present,
- Include A: feature A should be present, and
- Exclude A: feature A should not be present.

The first two kinds of constraints are called diagram constraints since they express fixed, inherent, dependencies between features in a diagram. The last two kinds of constraints are called user constraints since they express the user requirements regarding presence or absence of a feature. The user constraints may vary between subsequent uses of the feature diagram. The purpose of constraints is to further limit the variability of a feature diagram.

5.4.4 Sample of FDL

List 5-9 shows feature model of car in FDL [31]. A car consists of a carBody, Transmission, Engine and HorsePower.

```
Car: all( carBody, Transmission, Engine, HorsePower, pullsTrailer? )
Transmission: one-of( automatic, manual )
Engine: more-of( electric, gasoline )
HorsePower: one-of(lowPower, mediumPower, highPower)
```

List 5-9. Feature Expression for Car in FDL

5.5 Summary

This chapter provides a reference of WML, including CML, ADL and FDL, and illustrates them with small samples.

- WML that is unified with WSL makes it easy to bridge the gap between programming language and modelling language.
- WML provides a concise mechanism for describing the fundamental abstraction so that it can be applicable to handling complexity systems.
- WML supports self-documenting systems. Reversibility, combined with seamlessness, allows programs and models to be kept in synchronisation, and thus helps create and maintain system documentation.
- Definition of WML is close to the definition of UML so that WML can be mapped to UML easily.
- CML is defined based on OCL, UOL, POOSL, SDL and WSL. CML is a textual modelling language, which offers adequate modelling concepts.
- ADL is defined based on Acme, which can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.
- To get a better understanding of feature diagrams and enable creation of automatic tools for processing feature diagrams, a textual representation is preferable. Feature Description Language (FDL) is adopted as part of WML.

Chapter 6
Model Transformation Language

Objectives

- To define and classify the model transformation.
- To introduce the model transformation language, *Meta*WML.

This chapter aims at providing a reference of model transformation language, *Meta*WML. Firstly, model transformation is defined and classified. Secondly, the characteristics of *Meta*WML are analysed. Thirdly, main concepts and structure of *Meta*WML are described, and finally, the query facilities, action primitives and metric functions of *Meta*WML are presented.

6.1 Model Transformation

6.1.1 Classification of Model Transformation

A model transformation is a mapping of a set of models onto another set of models or onto themselves, which can be broken into two broad categories: model translation and model rephrasing. In the former, a model is transformed into a model of a different language, e.g. translations between WML and UML, and in the latter, a model is changed in same modelling language [145]. Many of these mapping activities are performed as automated processes that take one or more source models as input and produce one or more target models as output, while following a set of transformation rules.

Model transformation can be separated into vertical and horizontal dimensions, where the vertical dimension represents the different levels of abstraction in a particular software system and the horizontal dimension represents the different parts of a system on the same abstraction level. Mellor et al. break down the horizontal and vertical mapping processes into five types of model transformations, called: Refinement, Abstraction, Migration, Merge and Identification [103], which are special cases of model transformations that can be found in many areas of software engineering.

- Model Refinement is the process of vertically transforming that adds platform specific details to an abstract model. MDA approach which transforms CIM to PIM, and then to PSM is a typical model refinement.

- Model Abstraction is the process of extracting abstract information from a detailed model in an upwards directed way. Model Abstraction forms therefore the complement to Model Refinement.

- Model Migration transforms one certain representation of a system into another one on the same level of abstraction. Model Migration is a kind of Model Refactoring that a model is restructured so that it becomes easier to understand and maintain while still preserving its externally observable behaviour [5];

- Model Merge is the technique to combine individual models, seen as different views or aspects, to form a complete software system. Model merging is a part of the model weaving process.

- Model Identification shows only a part of a software system, identified by an applied model filter.

In respect of software re-engineering, model transformation is focused on model abstraction and refactoring.

6.1.2 Model Transformation Approach

There are many different approaches available for model transformation; some of these

include: relational/logic, functional, graph rewriting, generator/template-based and imperative [28]. Automated model transformation is based on the concepts of meta-modelling, which is similar to the program transformation that makes use of meta-programming techniques. To create a model transformation, which not only works for one specific model but for all the models of a certain modelling language, these transformations have to be based on the meta-models of the involved transformation items.

In this research, two modelling levels, model and metamodel level, are used to describe model transformation. The entities in metamodel level are further classified in two disjoint sets: modelling entities, such as classes and associations, represent the system structural information, and modelled entities, such as objects or links represent run-time information. Accordingly, there are also two kinds of instantiation relations. The relationship mapping modelling entity with its modelled entities is horizontal instantiation, while relationship representing the instantiation mechanism of the metalanguage is vertical instantiation, where horizontal instantiation preserves vertical instantiation [128].

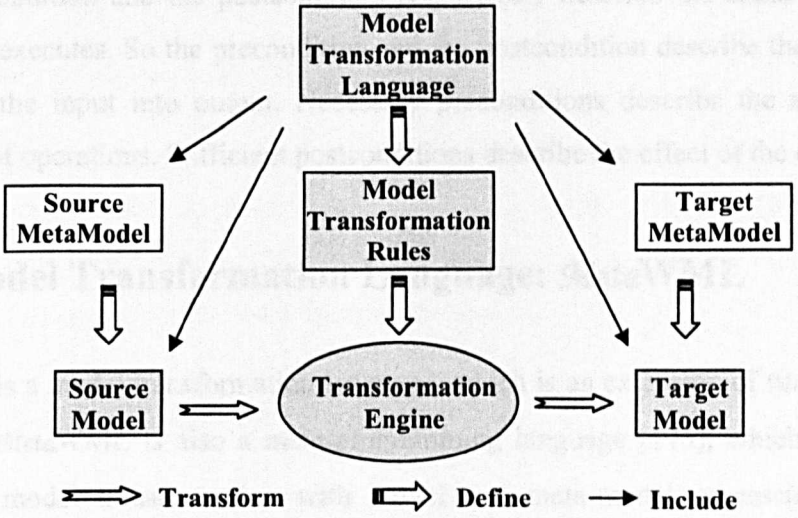


Figure 6-1. Overview of Model Transformation Approach

As shown in Figure 6-1, a *Source Model*, conforming to a *Source Metamodel*, is transformed into a *Target Model* that conforms to a *Target Metamodel*. The *Model*

Transformation Language makes it possible to specify *Model Transformation Rules*, which can be used by *Transformation Engine* to produce a *Target Model* from a *Source Model*.

6.1.3 Model Transformation Rules

A model transformation rule should contain the following information:

- Source and target language elements from their meta-models.
- The source end invariant stating the conditions that must hold in the source model for this transformation rule to apply. (Pre-condition)
- The target end invariant stating the conditions that must hold in the target model for this transformation rule to apply. (Post-condition)
- A set of mapping rules, each of which matches source model pattern and rewrites into target model.

The precondition and the postcondition respectively describe the status before and after a task executes. So the precondition and the postcondition describe the operations to transfer the input into output. Necessary preconditions describe the applicability conditions of operations. Sufficient postconditions describe the effect of the operations.

6.2 Model Transformation Language: *Meta*WML

*Meta*WML is a model transformation language, which is an extension of *Meta*WSL [87, 165, 180]. *Meta*WML is also a meta-programming language [175], which defines all aspects of model transformation with model and meta-model processing abilities. *Meta*WML gives the user a tool for analysing, rewriting, and simplifying both programs and models. Model transformation rules in Transformation Engine are implemented with this particular language. This section gives a brief reference to the possibilities of the *Meta*WML language. For a complete list of commands and functions, please refer to

appendix B.

6.2.1 Characteristics of *Meta*WML

The key to designing *Meta*WML is to offer model transformation abilities that can cover the largest possible range of situations. Besides having all the standard features of modelling language, the transformation language contains commands, functions and routines for operating on model elements. The following would be the characteristics for *Meta*WML.

6.2.1.1 Hybrid of Declarative and Imperative Language

The preferred style of transformation language is the declarative one: it enables to simply express mappings between the source and target model elements and hence can greatly simplify the description of transformation rules. Imperative languages, on the other hand, offer a command paradigm, namely sequence, selection, and iteration that are difficult to be expressed declaratively. In this direction, a language that mixes both kinds of approaches could demonstrate the advantages of both worlds [4].

6.2.1.2 Query Language for Pattern Match

A transformation is applied against certain model configurations. Thus, it would be desirable in many cases to describe the conditions under which the transformation produces a meaningful result, which can be treated as a pattern. For this purpose, a transformation language with query facilities is desired.

6.2.1.3 Supporting Composite Transformation

Because it is almost always easier to compose components than to build something from basic parts, it is often desirable to combine existing transformations to build new composite ones. Furthermore, it might be easier to build and test a transformation piecemeal by describing its parts first and then bringing them together to form the whole. So, a transformation language should support composite transformation.

6.2.2 Program Structure of *Meta*WML

*Meta*WML is a model transformation language that operates on model and meta-model level and composes of rules to define how source model elements are matched and navigated to create and initialise the elements of the target models. Since each layer in WML is formally defined with BNF, *Meta*WML can use the same mechanism as *Meta*WSL to manipulate WML elements which are presented with AST.

6.2.2.1 Name Convention

Since *Meta*WML is the extension of *Meta*WSL, they have the same naming convention defined by following rules:

- All *Meta*WML functions begin with the symbol "@".
- The first letter in each word of the name of a procedure or function begins with a capital letter.
- The words being separated by the underline character.
- Boolean function names have all an ending in "?".

6.2.2.2 Navigation on AST

To be able to navigate through the AST, *Meta*WML uses the same navigation mechanism as *Meta*WSL. The current model can be returned by the parameterless function @Model and the current item be returned by the function @I. The functions @Parent and @GParent return the parent and grandparent of the current model item. Initially the current item is the current model. Current position can be retrieved through @Posn, the current item can be defined like @I = @Model^@Posn. To travel now through the tree, *Meta*WML provides the commands @UP, @DOWN, @LEFT and @RIGHT or simply @GOTO if the desired position is known. To check if a step is possible the transformation program can use the commands @UP?, @DOWN?, etc.

6.2.2.3 Current Item Edition

*Meta*WML also provides a wide range of editing facilities to alter the model structure which can be presented via AST. These commands are considered to be the "heart and soul" of the whole transformation engine.

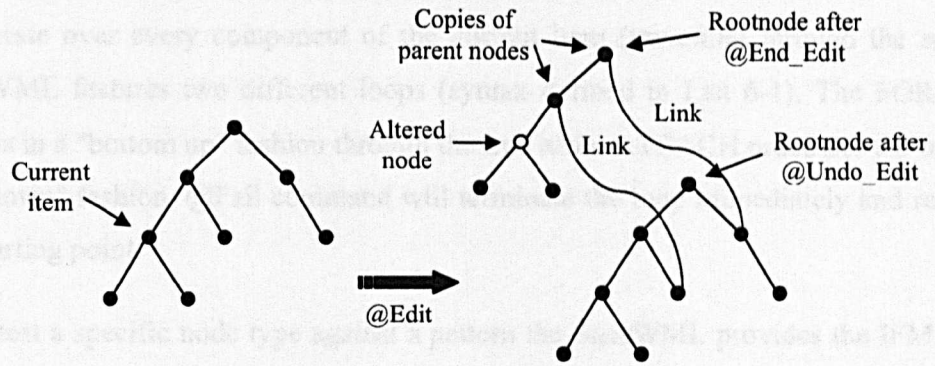


Figure 6-2. AST Before and After @Edit

As shown in Figure 6-2, the command @Edit can alter the current item as desired. Internally the transformation engine creates a copy of the current item, its subnodes and their parent nodes. All changes are now done to the copied nodes. Depending on the finalisation command (@Undo_Edit or @End_Edit) the current program pointer will hold the old root node or the copied one. If an error occurs the command @Undo_Edit can restore the original model, while an @End_Edit will commit the changes. This is a very powerful feature, providing rollback function. If something goes wrong during an alternation process, *Meta*WML can easily restore the original model.

New items can be inserted via the @Splice_Over(<Items>) overwriting the current item or with @Splice_Before(<Items>) and @Splice_After(<Items>) to the left or right of the current item. Items can be deleted with the @Delete command, which may result in syntax errors and probably an invalid @Posn position. @Clever_Delete command is provided that can delete the current item and "fix up" the syntax of the resulting model.

Transformation engine can also do copy & paste operations with buffer, which is represented by the @Buffer command. The buffer is filled with the @Cut command which deletes the current item and stores it in the buffer. The contents of the buffer can

be inserted again with @Paste_Over(@Buffer) overwriting the current item, with @Paste_Before(@Buffer) as new sibling to the left of the current item or with @Paste_After(@Buffer) as new sibling to the right of the current item.

6.2.2.4 Iteration Structure and Pattern Match

To iterate over every component of the current item (travelling through the subtree) *Meta*WML features two different loops (syntax defined in List 6-1). The FOREACH iterates in a "bottom up" fashion through the tree while ATEACH processes the tree in a "top down" fashion. @Fail command will terminate the loop immediately and return to the starting point.

To test a specific node type against a pattern the *Meta*WML provides the IFMATCH statement. This statement (syntax defined in List 6-1) does a pattern match on the current item. The following types of nodes can be visited with these loops: Statement, Statements, Terminal Statement, Terminal Statements, STS (short for Simple Terminal Statement), NAS (short for Non-Action System), Expression, Condition, Variable, Global Variable, Lvalue, Type_Decl and Relation.

```
FOREACH <type> DO
...
OD

ATEACH <type> DO
...
OD

IFMATCH type schema
THEN ...statements...
ELSE ...statements...
ENDMATCH
```

List 6-1. Syntax Definition for Iteration Structure and Pattern Match

6.2.3 Query Facilities of *Meta*WML

In order to manipulate the model elements (modelling entities and modelled entities), *Meta*WML defines the query facilities that enable to specify requests onto models. Query facilities are constructed on static language structure (AST), which are

implemented as *Meta*WML functions and organised in *Meta*WML library. The library is organised in a two-layer structure (Table 6-1):

- The basic layer contains the query facilities on the core concepts of modelling language, i.e. WML, such as classes, methods and associations. The basic layer provides all the basic mechanisms to query about the model elements.

- The architectural layer builds on basic layer and adds a lot of auxiliary functions to provide more complex queries on architectural notations. Design patterns and system aspects are focused on this layer. This layer raises the level of abstraction significantly otherwise lots of code have to be written.

Layer	Description
Basic	Query facilities on the core concepts of WML, such as classes, methods and associations.
Architectural	Auxiliary query facilities to provide more complex queries on architectural notations. Design patterns and system aspects are focused on this layer.

Table 6-1. Query Facilities of *Meta*WML in Different Layers

6.2.3.1 Query Facilities in Basic Layer

Query function typically involves the information retrieval of a group of model elements. Model elements are modelling entities, such as class, method, attribute, association, generalisation, aggregation, composition, realisation and dependency, and modelled entities, such as object, link.

Table 6-2 shows a list of query facilities in the basic layer. Each function includes its name, parameter types, returns type, and a brief textual description of what its purpose is.

Query Facilities	Return Types	Descriptions
@Class_Query	True/False	Judge whether class c is indeed a class

(Class c)		
@Class_Query (Class c, Method m)	True/False	Judge whether method m belongs to class c
@Class_Query (Class c, Attribute a)	True/False	Judge whether attribute a belongs to class c
@Class_Query()	Name List	Return all the class names in the system into a name list
@SuperClass_Query (Class c)	NameList	Return father names of a class (Return NULL if there is no father)
@SubClass_Query (Class c)	NameList	Return subclass names of a class (Return NULL if there is no child)
@Class_Substitutable (Class c1, c2)	True/False	Judge whether class c1 can be substituted by class c2
@IsAbstract (Class c)	True/False	Judge whether a class is declared to be abstract.
@Method_Query (Class c)	Name List	Return all the public method signatures in a class into a name list
@IsAbstract (Method m)	True/False	Judge whether a method is declared to be abstract.
@Attribute_Query (Class c)	Name List	Return all the public attribute names in a class into a name list
@InAssociation_Query (Class c)	Name List	Return role names of a class into a name list
@OutAssociations_Query (Class c)	Name List	Return role names of a class into a name list
@ExsitInAssociation (Class c)	True/False	Judge whether a role exits
@ExsitOutAssociation (Class c)	True/False	Judge whether a role exits
@Relationship (Class c1,c2)	Relationship List	Return all the relationship between two classes.
@Objs_Retrieve	Handle List	Return all the object handles of a class, including objects of subclass.

(Class c)		
@Link_Retrieve (Object o1, o2)	True/False	Judge whether there is a link between o1 and o2

Table 6-2. Query Facilities of *Meta*WML in Basic Layer

"Class c" means the parameter is a string that represents a class name. When the query function is invoked, only a string parameter is required. Name List is string list separated by comma.

Every query function is multi-way usable. For example, the Class_Query allows people to check if the passed argument is a class, or to query all classes.

The following query function asks whether the argument class c is indeed a class:

@Class_Query (Class c);

The following query function asks whether the argument Method (m) or Attributes (a) belong to a class:

@Class_Query (Class c, Method m);
@Class_Query (Class c, Attributes a);

A query asking for all the classes in the system can also be performed without the argument:

@Class_Query ();

There are also query functions for modelled entities, for example:

obj_list = @Objs_Retrieve (Class c)

retrieves all instances of the specified class type that exist at any one time and assigns a reference to the instances to obj_list. The obj_list is the returned reference to the set of instances of the specified class.

6.2.3.2 Query Facilities in Architectural Layer

With the basic layer query functions, more complex query functions could be added. This allows people to perform more complex queries without writing the query code. These query functions can be used to query at a high-level of abstraction.

Query facilities in architectural layer are based on architectural knowledge. Patterns and Aspects are both structural knowledge of software architecture. The implementation of design pattern detection and joinpoint selection are discussed in Chapter 7. Since this research is still in its initial stage, only a small group of design patterns have been investigated.

Table 6-3 illustrates Query facilities in architectural layer with an example of CompositePattern design pattern, in which relationship between two classes is used in 4 different ways:

Query Facilities	**Return Types**	**Descriptions**
@ CompositePattern (Class c1, c2)	True/False	Judge whether CompositePattern relationship is hold for class c1 and c2
@ CompositePattern (Class component)	Name List	Return all the composite classes into a name list
@ CompositePattern (Class composite)	Name List	Return all the component classes into a name list
@ CompositePattern	Name List	Return all the classes in CompositePattern into a name list

Table 6-3. Query Facilities of Composite Pattern in Architectural Layer

- When two actual classes are passed, @CompositePattern check whether these two classes are in a composite pattern relationship.
- When only the component class is passed, all the classes that play the role of composite class will be returned.
- When composite class is passed, all the component classes for that composite class will be returned.
- When no information is passed, all possible component and composite classes will be returned.

6.2.4 Action Primitives of *Meta*WML

The ability to manipulate WML depends upon the capability to add, delete, and connect model elements within the model. *Meta*WML provides a minimal set of actions for expressing behaviour. An action is a "fundamental unit of behaviour specification that represents the transformation or processing in the modelled system" [122]. Action primitives are used to specify imperative logic in a form that is constructed on static language structure, AST.

Table 6-4 lists the action primitives of *Meta*WML:

Action Primitives	Return Types	Descriptions
@Add_Class (Class c)	N/A	Add a class c
@Remove_Class (Class c)	N/A	Remove a class c
@Extract_Class (Class c1, c2)	N/A	move code to a new class
@Add_Method (Class c, Method m)	N/A	Add a method to class c
@Remove_Method (Class c, Method m)	N/A	Remove a method from class c
@Move_Method (Class c1,c2, Method m)	N/A	Move a method from one class to another
@Extract_Method (Class c1,c2, Method m)	N/A	move code to a new method
@PullUp_Method (Class c, Method m)	N/A	Move a method from subclass to a superclass
@Create_Instance (Class c)	Object Handle	Create an object from class c
@Destroy_Instance (Object obj)	N/A	Delete an object.
@Create_Link	N/A	Create a link that complies with an

(Object obj1, obj2)		association.
@Destroy_Link (Object obj1, obj2)	N/A	Delete a link between tow objects.
@Destroy_Link (AssociationRole r)	N/A	Delete a link based on an association role.

Table 6-4. Action Primitives of *Meta*WML

6.2.4.1 Primitives on Modelling Entities

Model transformation typically involves the addition or removal of a group of modelling entities, such as class, method, attribute, association, generalisation, aggregation, composition, realisation, and dependency.

The transformation program written in *Meta*WML takes the input and performs the basic transformations include the addition or removal of a modelling element. The replacement of a model element with another is conducted by first removing the modelling element and then adding a new modelling element. The output is the transformed WML model.

6.2.4.2 Primitives on Modelled Entities

Model transformation typically also involves the creation or destroying of a group of modelled entities, such as processes, objects and links. These basic transformations become the building blocks of dynamic evolution.

The concrete syntax for creating instances of objects is:

```
obj := @Create_Instance(Class c);
```

This command creates instances of a class and then returns a reference of the instance to obj.

The concrete syntax for deleting instances of objects is:

```
@Destroy_Instance (Object obj);
```

This construct removes the instance of the object referenced by obj. The construct

also deletes any links connected to the obj.

The create link construct creates a link between two model elements. The concrete syntax is:

@Create_Link(Object obj1, obj2);

The handles, obj1 and obj2, are references to instances of the model elements that are connected together. The end of the link connected to obj1 is the source of the association and the link end connected to obj2 end is the target.

The destroy link construct deletes the link that exist between two model elements. The destroy link construct has two definitions for its concrete syntax. The first definition is:

@Destroy_Link (Object obj1, bj2);

The handles, obj1 and obj2, are references to instances of the model elements which are connected to the link to be deleted.

The other definition deletes all references to the AssociationRole specified by the association rolename. The concrete syntax is defined as:

@Destroy_Link(AssociationRole r);

The parameter AssociationRole specifies the role that is played by the link to be deleted. When an instance of an AssocationRole is removed, the association is no longer available to the domain in which it was defined.

6.2.5 Metric Functions of *Meta*WML

Software metrics are a key technology for managing reverse engineering projects. Well-developed software metrics for reverse engineering will be a great aid to software engineers [180]. There are three classes of software properties whose attributes should be measured: processes are collections of software-related activities; products are any artifacts, deliverables, or documents that result from a process activity; resources are entities required by a process activity [40]. In reverse engineering, it is mainly the product attributes that will be measured. The products in reverse engineering are

existing systems.

Both program transformation and model transformation uses transformation rules that have source and target patterns. Metrics on software transformation are useful to develop heuristics approach on less complex, more structural final results. To see how good or bad a transformation performs, MetaWML provides a number of metric functions which measure the complexity and abstractness of a given WML item. Metric functions can be treated as a kind of query facilities, which return the properties of program or model.

6.2.5.1 Metrics for Program Transformation

Metric functions for program transformation are focused on WSL level. Following metric functions have been used [87, 180]:

@Stat_Types(I): Return set of statement types appearing in I
@Total_Size(I): Total number of nodes (items) in I
@Stat_Count(I): Total number of statement items
@Gen_Type_Count(type, I): Number of occurrences of given generic type
@Spec_Type_Count(type, I): Ditto for a specific type
@McCabe(I): McCabe cyclometric complexity measure for I
@CFDF_Metric(I): Control-flow / data-flow metric for I
@BL_Metric(I): Branch-loop metric for I
@Struct_Metric(I): A weighted sum over all the items in I

When type of I is class, then:

@ WMC (I): Return weighted methods per class in I
@ DIT (I): Return depth of inheritance tree in I
@ NOC (I): Return number of children in I
@ NVC (I): Return number of variables per class in I
@ APM (I): Return average parameters per method in I
@ NOO (I): Return number of objects in I

Metrics on programming languages provide only one way to measure the system. Models provide more abstraction information on system structure, measuring the various models can bring insight into these models.

6.2.5.2 Metrics for Model Transformation

Model metrics can be simple or complex. A simple metrics like @TNC (Total Number of Classes) is a primitive metric needed for the complex metrics. As examples of complex metric, two metrics for measuring AOP model are defined.

Assuming:

m = the amount of target classes,

k = the amount of Joinpoints in one target class,

l = the total line number of the source code, which is used to invoke the Aspect function,

s = the total line number of the source code of Aspect function,

δ = the total line number of the source code of Aspect base-class,

$Count_i$ = the amount of public member functions of the *ith* target class, then

the code conciseness rate, C_r, can be computed as: $C_r = \frac{m \times k \times l \times s}{m + s + \delta}$, and

the system efficiency rate, E_r, can be computed as: $E_r = \frac{1}{\sum_{i=1}^{m} k \times Count_i}$.

6.2.5.3 Examples of Model Metrics

Two examples are selected to depict model metrics, one called SQL Verification and another one called Event Logger. The SQL Verification shows how to add a new function to a few classes, while the Event Logger shows how to add a new function to all or most of the classes.

Table 6-5 is a general analysis of above two cases. In the first case, code conciseness rate is 1.38 and the second one is 6.18. In [80], Kiczales showed that this value could

reach to 98, which is much bigger than the results of this thesis. In order to judge whether the member function is a Joinpoint, all the target classes in the first case need 439 times additional judgement at runtime and in the second need 5380 times, which leads to a great descending of system efficiency. The efficiency of evolved system can still be accepted, since both of their efficiency rates are higher than the low limitation, 0.0001. From Table 6-5, it can be seen that the amount of interface checks decreases 38 times for the first case and 363 times for the second.

Comparing Items	**SQL Verification**	**Event Logger**
Amount of target classes	39	64
Amount of Joinpoints in one target class	1	6
Code line number without AOP	408	1824
Code line number with AOP	296	295
Rate of code conciseness	1.38	6.18
Decrease of interface checks	38	363
Additional judgements	439	5380
Efficiency rate	0.0023	0.0002

Table 6-5. Performance Analysis

Many factors will influence the efficiency of a system, such as the amount of Aspect classes, the amount of the target classes and the amount of the public member functions of each class. It is obvious that applying Aspect classes will lengthen the message chain and lead to additional checks. The experiment (Figure 6-3) shows that the efficiency rate decreases rapidly before the amount of target classes reaches 20 and it remains almost constant after the amount of target classes reaches 48. It means that, after the scale of the target classes reaches to a high point, the efficiency rate will rely mainly on the efficiency of Aspect function than the amount of target classes.

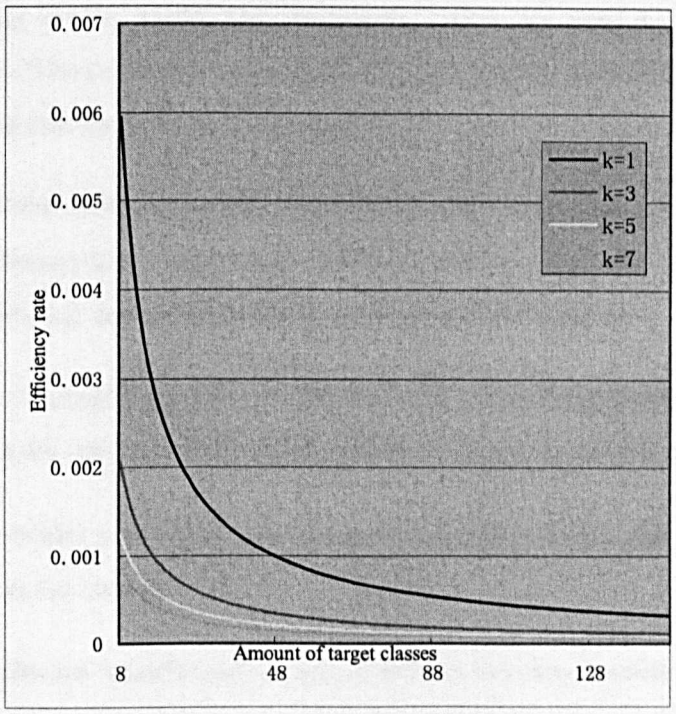


Figure 6-3. Efficiency Rate as a Function of the Amount of Target Classes for Different k

6.3 Summary

In this chapter, the classification and architecture of model transformation are introduced and a reference of model transformation language, *Meta*WML, is provided.

- A model transformation is a mapping of a set of models onto another set of models or onto themselves following a set of transformation rules. Model translation transforms a model into different modelling language, while model rephrasing changes in same modelling language.

- Model transformation can be separated into vertical and horizontal dimensions, which can be broken down into five types of model transformations, called: Refinement, Abstraction, Migration, Merge and Identification. In respect of software re-engineering, model abstraction and refactoring are focused.

- Two modelling levels, model and metamodel level, are used to describe model transformation. The entities in metamodel level are further classified in two disjoint sets: modelling entities and modelled entities.

- A model transformation language *Meta*WML is the extension of *Meta*WSL with model and meta-model processing abilities, which gives the user a tool for analysing, rewriting, and simplifying both programs and models.

- Query facility can manipulate the model elements (modelling entities and modelled entities) to specify requests onto models, which is based on pattern matching.

- *Meta*WML provides a minimal set of actions to add, delete, and connect model elements within the model.

- Metrics on software transformation are useful to develop heuristics approach on less complex, more structural final results. *Meta*WML provides a number of metric functions which measure the complexity and abstractness of a given WML item.

Chapter 7
Implementation of Model Construction and Transformation

Objectives

- To illustrate the model construction from WSL.
- To illustrate the model transformation for abstraction.
- To illustrate the model transformation for refactoring based on design patterns and aspects.
- To define the model transformation between WML and UML.

This chapter focuses on the algorithm and implementation of REMOST approach based on WML and *Meta*WML, in which model construction and model transformation are two main concerns. Firstly, model construction from WSL is examined. Secondly, Model transformation for abstraction and model transformation for refactoring are investigated, and finally, unifying UML and WML is discussed.

7.1 Model Construction from WSL

Model construction from source code of legacy system is the bridging process that transforms a program into a model. In this research, a generic mechanism that can generate a specific bridge between grammarware and modelware [173] based on the BNF of WSL and CML is proposed. Bridging programware and modelware involves

several tasks, such as processing the artifacts in programware and transforming them into modelware. Manual bridging is an exhausting and recurring task, which needs tool support for automatic transformation between WSL and CML.

ADM [116] initiated by OMG has been introduced in Chapter 3. The main target of ADM is to rebuild existing applications as models and then perform refactoring of the models or transform them to new target architectures. The proposed transformation is similar to ADM task, which consists three major phases:

- The first phase focuses on representing the WSL code in the terms of an Abstract Syntax Tree (AST).
- The second phase aims at analysing the software entities, such as data types, variables, functions and parameters.
- The third phase extracts CML models by an incremental clustering algorithm.

The CML model is an object oriented model. In order to obtain the CML model from its legacy source code, in most cases, object class identification methods are developed to transform a subject system from its original procedural language implementation to an object oriented design model. Migrating legacy software systems to object oriented platforms has received significant attention over the past few years. Software re-engineering community has already proposed a number of different methods to migrate procedural code into object oriented platforms. These methods include concept analysis, cluster analysis, slicing, data flow and control flow analysis, source code features, and informal information analysis [185]. However, no matter how sophisticated the analysis techniques are, user assistance and guidance is crucial on obtaining a viable and efficient object model. The user in order to guide the discovery process and to obtain a better and a more suitable object model can also utilise domain specific information. In the following sections, the model construction method, which restructures procedural oriented system into object oriented system, is outlined based on current work in SERG [15, 97, 105, 131, 180].

7.1.1 Parsing and Analysing of WSL Program

The first step in the proposed transformation is to parse the grammar of WSL, which has been fully implemented in previous research [163, 165]. Since both WSL and CML are defined by BNF grammar, the transformation rules are organised along the major BNF concepts.

For software reverse engineering, it can be used to thoroughly describe an existing system in terms of entities and their relationships. In order to construct model from program, it is necessary to refer to various program elements: classes, methods, interface, argument, object reference, field, parameter, expression, variable, and method invocation.

Relationships between modules can tell people something about cohesion and coupling of modules, or about the layers built into the legacy system. If one procedure invokes many others (high fan-out), and doesn't get invoked itself, it is likely to be a control (coordination) module, with little built-in functionality. Likewise, if a procedure is called by many others (high fan-in), it is likely to be some sort of utility routine, dealing with error handling or logging. The procedures with both low fan-in and low fan-out are the ones that are likely to contain business logic.

Slicing techniques can be used for analysing module relationships, e.g. for each output a backwards slice is computed: this slice contains all the code needed to compute this output of the module. Overlapping slices can be factored out into shared subroutines.

Outlined below presents a process used to parse and statically analyse the WSL source code to determine program elements and their relationships. All the globe variables, functions and parameters and their relationships in the original procedural system are identified and stored in Repository (List 7-1).

```
STARTDCT
  COUNT 11
  MODULE FMT001A0
    FUNCTIONID 5
      STARTSECTION
        SECTIONID 1
```

```
        DCDNAME F001A0C1
        STARTDATA
      ELEMENT FMT001A0_CSECT
        DATAID 1
        TYPE 1
        PARENT 0
        MORE
      ELEMENT R3
        DATAID 2
        TYPE 5
        PARENT 1
        MORE
        ...
    ENDDATA
  ENDSECTION
ENDDCT

STARTFCT
  COUNT 65
  MODULE PROJ01
    FUNCTIONID 1
    TYPE 0
    EXECLINES 719
    COMPLEXITY D
    MCCABE 85
    PARENT 0
    VERMAJ -1
    VERMIN 0
    MORE
    MODULE MODULE
      FUNCTIONID 2
      TYPE 1
      EXECLINES 612
      COMPLEXITY D
      MCCABE 79
      PARENT 1
      VERMAJ -1
      VERMIN 0
      MORE
      ...
ENDFCT
```

List 7-1. An Example of Data and Control Informaiton in Repository

7.1.2 CML Model Extraction Algorithm

The key point in transforming a procedurally structured legacy system to an object-oriented system is to identify possible object classes within the legacy source code and then restructure the procedures and variables of the old system into methods and attributes respectively of classes of the new system.

Due to the object oriented design principle that a class encapsulates data and related methods, relations between data declarations and functions are focused. Such relations include type references, data updates, and data uses. Hence, there are two aspects for object class identification: one aspect (function-driven) uses legacy functionality as its primary basis for class extraction, the other aspect (data-driven) searches for persistent data elements as the basis for its class extraction. Global variables and their data types in the original legacy source code become primary candidates for classes in the new object oriented system. Similarly, functions and procedures in the original system become primary candidates for methods and are attached to the aforementioned identified classes. Furthermore, the object-oriented model is refined by the identification of association and aggregations for the new migrant systems.

Clustering techniques utilise certain criteria to decompose a system into a set of meaningful modular clusters. Such criteria attempt to achieve a cluster with low coupling, high cohesion, interface minimisation and sharing of neighbour resources. The decomposition of a program produces a set of smaller clusters, which contain the ASTs of segments of the procedural code to be migrated. To facilitate object-oriented model extraction, a set of criteria are identified, which aim at achieving high encapsulation, high cohesion within a class, and low coupling between classes.

- C1: If two variables share a common data dependency, these two variables should probably be assigned to the same object class.
- C2: If two procedures have a high degree of coupling, or interaction among themselves, these two procedures probably should be assigned to the same object class.
- C3: Procedures with a high fan-out are usually control modules and thus should be kept in a separate control class.
- C4: Procedures with a high fan-in are usually log modules and thus should be kept in a separate log class.

Once the object classes are identified, it is necessary to restructure the procedural

representation into an object-oriented one. This involves encapsulated related procedures and their variables into a class structure. After the encapsulation of procedures and variables within class structures has completed, the parameter list of each encapsulated procedure is modified to reflect this class structure. If a method accesses an attribute of its own class, this attribute does not need to be passed in as a parameter to the method. Instead, this attribute can be referenced directly within this method.

List 7-2 presents an algorithm used to determine the degree of coupling between the codes' various procedures and variables and partially based on this coupling, cluster closely-coupled variables and procedures into their relevant classes. Incremental process is applied to cluster the maximal size of source code entities into classes.

Algorithm: CML Model Extraction

Begin:

-- Initialise a set of Classes, Methods and Attributes.
Ci = Filename; -- Each file is treated as a class to set up initial classes set
Ac = Variblenames -- Each varible in a file is treated as an attribute
Mc = Procedurenames – Each procedure in a file is treated as a method

-- Collect Information of Varibles and Procedures Collect.

-- Determine procedure by the <Proc> "ProcName" ... <End> declaration structure.
-- Determine the Level of the procedures and varibles.
-- Determine the usage of the parameters.
Build VarList = (variable name, procedure name, level, usage of variable, usage)

-- Determine the usage of the Caller.
-- Determine the usage of the Called.
Build ProcList = (procedure name, level, usage of caller, usage of caller)

--Sort the VarList, ProcList by Usage descending.
Sort VarList
Sort ProcList

-- Choose the possible cluster elements

-- Define the threshold
$\theta = 0.9$ --modifiable
-- procedures that are called by other procedures but call no procedures themselves
Find LogCluster
-- procedures that call other procedures but call no procedures themselves
Find ControllerCluster

```
    -- These procedures are placed in separate classes respectively. MetaWML is used.
    Create NewClass
    -- Procedures and Variables are removed. MetaWML is used
    Remove Procedure & Varibles

End
```

List 7-2. CML Model Extraction Algorithm

7.2 Model Transformation with *Meta*WML

From the technical point of view, model transformation in reverse engineering relies on the concept and description of abstraction, which is an effective way to reduce the complexity of software systems. Abstraction is a process that transforms lower-level elements into higher-level elements containing fewer details on a larger granularity. Abstraction is the crucial technique to reverse engineering. Without tackling abstractions properly, any design or specification recovery methodology can not succeed.

Transformation techniques can ensure consistency and provide a reliable linkage between the various stages in system development. Once a linkage at two different abstraction levels is established, it permits a rapid re-engineering in response to changes. So the major concerns of abstraction are to build or recover various relationships at different abstraction levels.

There are potentially many logical ways of decomposing a system, which implies that there are many ways of abstracting a system. In order to jump from one level up to another abstract level in the process of reverse engineering. One has to throw away some information. No method can guarantee that such a throwing away of information is appropriate. This implies that the abstraction is creative work. In order to achieve correct and practical abstraction, human interaction and knowledge base for transformation engine are necessary.

7.2.1 Model Abstraction

Models can represent several levels of abstraction in terms of model elements and their relationships. This knowledge can be used to define the model abstraction rules. Abstraction rules have input and result patterns. An abstraction algorithm can performs syntactic matching of the abstraction rules on the model. Whenever an input pattern of a rule is encountered in the model, then that pattern is replaced by the result pattern of that rule. Since every abstraction rule has a result pattern that is simpler and more abstract than its input pattern. It follows that every application of a rule simplifies a given model [34].

Model abstraction rules presented in this section are focused on class abstraction, which are far from enough for every aspects of reverse engineering. However, the main purpose is to propose an approach to rule definition and shows the possibility of the approach. All kinds of abstraction rules and the methods used to define abstraction rules is based on the work of [34, 36]. The presented abstraction rules are generic and applicable to a wide range of software projects.

Class abstraction has a number of vital uses [34]: (1) it aids program and model understanding by reducing the number of lower-level elements to the most important, higher-level elements; (2) it supports consistency validation by comparing existing higher-level models or architectures with abstracted ones; (3) it assists reverse engineering by transforming lower-level models into higher-level ones. Class abstraction can be achieved by having the focus on the ability to identify key structural relationships between classes. Currently four types of key structural relationships are supported: generalisation (inheritance), association (calling direction), aggregation (part-of), and dependency (uses or interfaces). Considering directionality, this implies eight unidirectional relationship types such as GeneralisationRight or AggregationLeft plus three bidirectional relationship types Association, [Agg]Assocition, and Association[Agg]. Altogether, those relationships can form 121 different patterns (11*11). Some of those patterns (92 patterns) are abstractable while other patterns (29 patterns) are not abstractable. Given that it should not matter from what direction a pattern is viewed (or abstracted), it implies that mirror images of abstraction patterns

must have the same values. Due to the limit of space, Table 7-1 only gives a list of abstraction pattern rules for GeneralisationRight with 11 relationships: GeneralisationLeft, GeneralisationRight, DependencyLeft, DependencyRight, AssociationLeft, AssociationRight, AssociationLeft[Agg], [Agg]AssociationRight, Association, [Agg]Association, Association[Agg].

No.	Input Pattern	Result Pattern
1.	GeneralisationRight - Class – GeneralisationRight	GeneralisationRight
2.	GeneralisationRight - Class - DependencyRight	DependencyRight
3.	GeneralisationRight - Class - AssociationRight	AssociationRight
4.	GeneralisationRight - Class - [Agg]AssociationRight	[Agg]Association Right
5.	GeneralisationRight - Class - GeneralisationLeft	Φ
6.	GeneralisationRight - Class - DependencyLeft	DependencyLeft
7.	GeneralisationRight - Class – AssociationLeft	AssociationLeft
8.	GeneralisationRight - Class – AssociationLeft[Agg]	AssociationLeft[Agg]
9.	GeneralisationRight - Class – Association	Association
10.	GeneralisationRight - Class - [Agg] Association	[Agg] Association
11.	GeneralisationRight - Class - Association[Agg]	Association[Agg]

Table 7-1. Class Abstraction Rules for GeneralisationRight

7.2.2 Model Refactoring

The re-engineering processes generally focus on the increased quality of the systems. Software quality is defined as a set of features and characteristics of a software product that relate to external attributes, such as performance, and internal attributes such as, the complexity of data structures. Software quality properties reflect also the degree of the conformance to specific non-functional requirements [185]. Software quality can be measured by a collection of appropriate metrics.

The mapping between models established by the transformation is required to be preserved over time. Refactoring is an act of performing behaviour-preserving transformations. Behaviour-preserving transformations change the structure of models, without modifying their observable behaviour. They are used to keep behaviour views

and architectural views consistent or to integrate these views consistently into a unified model. Re-engineering of legacy system into 'new improved architecture' based system is a challenge [53]. The need is to identify places in the software architecture where quality would be improved by the introduction of design patterns and AOP.

Design patterns describe micro-architectures that solve recurrent architectural problems. It is important to identify these micro-architectures during the maintenance of object-oriented programs. The aspect-based technical approach can also help to reach the new improved architecture. The goal of AOP is to extract functionality that is scattered throughout whole application such as business rules, transactions, logging, errors, user interface and SQL operations into separate modules. This can greatly enhance software reuse, increase code modularity and reduce code tangling.

There is a natural relation between patterns and refactorings. Design patterns capture many of the structures and result from refactoring. Design patterns thus provide targets for the refactorings [47]. In this section, design patterns are examined for software restructuring, which focuses on providing a catalogue of model transformations to refactor existing system by the utilisation of design patterns. The emphasis is on to use design patterns to improve source code quality rather than to produce directly design pattern compliant source code [78]. Query facilities and action primitives defined in *Meta*WML are used to manipulate model elements for design pattern based model transformation and metric functions are used to measure the results, which includes two levels: one level is how to refactor towards a pattern to improve the current design and another level is how to manipulate the design patterns for further software evolution.

Towards design pattern transformation means that design patterns are investigated as a means to restructure a legacy system so that the new system conforms to specific design patterns and meets specific Non-Functional Requirement (NFR) criteria. A NFR denotes a feature of a system that is not covered by its functional description that typically addresses aspects related to the reliability, compatibility, ease of maintenance of a software system, and so on [154]. Refactoring towards design patterns requires dealing with model elements and their structural relationships in legacy system. Once the group of model elements and their structural relationships are identified in a given

model, a number of algorithms are applied to map these classes and their structural relationships to the most appropriate design pattern.

In [23], a method to automate the transformation of design patterns into existing code using transformation algorithms is introduced, in which a "precursor" indicates where a transformation begins (i.e., the starting point) and the design pattern serves as the target of the transformation, such that the transformation algorithm stop executing when the design pattern has been applied to the code. In [12], an approach is proposed to use class hierarchy, aggregation/association and message flow for design pattern detection. The proposed research combines above two approaches and provides a solution to detect design patterns in WML models. In general, a design pattern is detectable if its template solution is both distinctive and unambiguous [12]. Every pattern in the classic book Design Pattern [47] contains a structure diagram [78]. Structural components are examined as a first step to identify candidate patterns. Design information presented with WML, such as class hierarchy and relationship or the object model, can be used in the search for design patterns artifacts.

After a design pattern is applied in a software application, it could still be changed in the particular ways directed by the design pattern. The evolution information of each design pattern allows changing the system design with minimum impact of other parts of the system. As described in [33], five kinds of pattern-level transformations (Table 7-2) are recurring in different design patterns, which can be easily implemented by *Meta*WML.

No.	Transformation Names	Description
1.	Independent	Addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern.
2.	Packaged	Addition or removal of one independent class with attributes and/or operations and the corresponding relationships between this class and the classes in the original pattern.
3.	Class group	Addition or removal of one attribute/operation in several different classes consistently.

4.	Correlated classes	Addition or removal of a group of correlated classes.
5.	Correlated attributes/operations	Addition or removal of a group of classes and addition or removal of some attributes or operations in the classes of the original pattern applications.

Table 7-2. Summary of Pattern-Level Transformation [33]

7.3 Usage of *Meta*WML for Model Transformaiton

This section will investigate how to define and use model transformation in *Meta*WML. *Meta*WML query facilities and action primitives can perform queries on the model and replace the matched pattern with action primitives.

The first example shows how the abstraction rules defined in Table 7-1 can be implemented with *Meta*WML query facilities and action primitives that an abstraction algorithm written in *Meta*WML can perform queries on the model and replace the matched pattern with action primitives. List 7-3 shows a procedure that absorbs all association classes and abstracts complex class structures into a bigger picture:

```
proc @AbsorbAssociationClasses CMLCode(Data)
FOREACH DECLARATION DO
if @Spec Type(@Item) = ClassSignature
    if @ExsitInAssociation(@Item) and @ExsitOutAssociation(@Item)
    then @Absorb(@Item)
od.
```

List 7-3. A Procedure Using Abstraction Rules

In this example, a high-level *Meta*WML construct, the FOREACH construct, is used to iterate over all those components of the currently selected item which satisfy certain conditions, and apply various program/model transformation operations to them. This example shows that *Meta*WML enables a programmer to write complex program/model transformations in a few lines of code, leaving the system to deal with most of the details and the tricky special cases.

The second example is illustrated with a visitor design pattern. The general idea of

the visitor design pattern is to separate the structure of elements from the operations that can be applied on these elements. This separation makes it easier and cost-effective to add new operations, because the classes of the object structure do not have to be changed. The typical example of the visitor design pattern is to separate parse trees from the operations that are typically performed on these parse trees (such as generating code, pretty printing or optimisations) [175].

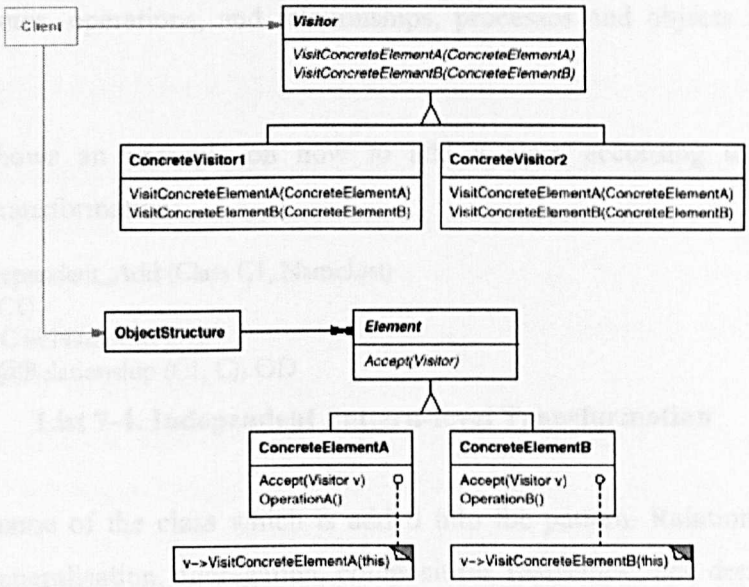


Figure 7-1. Structure Diagram of Visitor Design Pattern [47]

As depicted in Figure 7-1 [47], there is a hierarchy describing the elements, and there is a separate hierarchy implementing the operations. *Element* class is the root class of a hierarchy on which the class *Visitor* and its subclasses define operations. Every *Element* class defines a method *Accept* that takes a *Visitor* as argument and calls this *Visitor* using an operation that indicates its type. To detect a visitor design pattern, the rule describing the structure of the visitor design pattern is fairly straightforward. It expresses that the *Visitor* is an abstract class, and that it implements the visit method. In the same way, *Element* is an abstract class too, and implements methods called *Accept* with a *Visitor* as argument [175]. To apply this detection rule, Query facilities can be used to find the candidates that comply with the visitor design pattern. Since there are many ways to implement a pattern hence matching criteria for the design patterns

identification can not be formalised. A knowledge base for design patterns is needed and the user intervention is really important.

The third example is about the pattern directed transformation. The transformation program takes the input and performs the design pattern directed transformation according to the type of pattern-level transformation, which defines how to add or remove a group of model elements (modelling entities and modelled entities), such as classes, attributes, operations, and relationships, processes and objects into a design pattern.

List 7-4 shows an example on how to add a class according to Independent pattern-level transformation:

```
proc @ Independent_Add (Class C1, NameList)
  @Add ( C1)
  WHILE C in NameList DO
  @Add ( @Relationship (C1, C)) OD
```

List 7-4. Independent Pattern-level Transformation

C1 is the name of the class which is added into the pattern. Relationship includes association, generalisation, aggregation, composition, realisation, and dependency. The NameList is the existing class names from the original pattern. This kind of transformation appears in several design patterns, for example, in the Mediator and Facade patterns.

7.4 Unifying WML and UML

MDA is a technique that is based on UML and MOF. Since UML is a well-established industry standard, well-known by many industry developers and with good tool support. The system, as represented by UML diagrams, has a better chance of being properly understood by industry developers than a formal notation. In order to reuse and integrate with MDA environment, WML should be aligned with UML diagrams which can be presented visually and translated into XMI for information exchange and hence be supported by many tools.

In this section, the solutions to realising the alignment of WML with UML are described, which combine the advantages of intuitive graphical notations and formally defined textual notations. A semi-automatic translation method is defined to systematically create several models expressed in UML from WML. By translating WML notation to UML specification, there is assurance that this notation is UML-compatible and hence, this notation is easily convertible to the various UML exchange formats used by different UML modelling tools.

7.4.1 Mapping between CML Constructs and UML Components

There is an increasing need for methods and tools that can support model analysis. CML is designed as a textual notation to enable the creation of automatic tools. The purpose of mappings between CML and UML structure is two-fold. One is that, through CML, there is a direct relation between the original source code, the model of the restructured system, and the re-engineered target system. The other is that, to make CML UML-compliant, CML structures can be exported, via XMI, to selected visual UML tools. In this way, original source code can be first extracted into CML, then converted into XMI and imported into a UML visual modelling tool that produces visual UML diagrams of the system.

Since CML is defined based on UML/OCL, the CML Constructs and UML diagrammatic notation are syntactically similar. The CML can hence to represent UML diagram constructs such as UML's classes, associations, and activities. Each CML construct that represents a UML component, such as an activity, has a unique name. A diagram, such as an activity diagram, consists of a graph of these constructs linked together by their unique names. The CML constructs form the most-commonly-used elements within the UML components. In reengineering, there is often no need to use the Extension Mechanisms of UML to represent a system. Typically, one would try and confine the reengineered and re-documented system within the common and already-defined set of diagrams. Consequently, it is easy to map CML constructs to various UML modelling notations and vice versa.

Previous example in List 5-4 is used to illustrate the mapping from CML to UML.

Each package consists of a set of classes. Features in CML include both attribute and method. The relationship in CML can be translated into Association and AssociationEnd. The AssociationEnd structure has the fields of Role name, Class name, Multiplicity, an aagegration type and navigability. Multiplicity may be a one-to-one [1..1], one-to-many [1..*], zero-to-one[0..1], zero-to-many [0..*], and many-to-many [*..*]. As a result, CML can be translated into XMI without the position information (List 7-5).

```
<?xml version="1.0" standalone ="yes"?>
<XMI xmi.version="1.1" xmlns:UML="omg.org/UML/1.4">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>ru.novosoft .uml.impl .UMLRepositoryImplXMIWriter
      </XMI.exporter>
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <UML:Package xmi.id="a1" isRoot="true" isLeaf=" false "
      isAbstract =" false " name="car_production" isSpecification =" false ">
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id="a20" isRoot="false" isLeaf=" false "
          isAbstract =" true " name="CarType"
          isSpecification =" false " isActive =" false ">
          <UML:Namespace.ownedElement>
            <UML:Attribute xmi.id="a21" name="plate"
              isSpecification =" false " type="a2"></UML:Attribute>
            <UML:Attribute xmi.id="a22" name="model"
              isSpecification =" false " type="a2"></UML:Attribute>
            <UML:Attribute xmi.id="a23" name="trademark"
              isSpecification =" false " type="a2"></UML:Attribute>
            <UML:Attribute xmi.id="a24" name="maxSpeed"
              isSpecification =" false " type="a2"></UML:Attribute>
            <UML:Attribute xmi.id="a25" name="numberSeat"
              isSpecification =" false " type="a2"></UML:Attribute>
          </UML:Namespace.ownedElement>
          <UML:Classifier. feature >
            <UML:Operation xmi.id="a26" isRoot="false" isLeaf=" false "
```

```
        isAbstract =" false " isQuery=" false " name="Make"
        visibility ="public" isSpecification =" false ">
      </UML:Operation>
      <UML:Operation xmi.id="a26" isRoot="false" isLeaf=" false "
        isAbstract =" false " isQuery=" false " name="SetModel"
        visibility ="public" isSpecification =" false ">
      </UML:Operation>
      <UML:Operation xmi.id="a26" isRoot="false" isLeaf=" false "
        isAbstract =" false " isQuery=" false " name="GetModel"
        visibility ="public" isSpecification =" false ">
      </UML:Operation>
    </UML:Classifier . feature >
  </UML:Class>
  <UML:Class xmi.id="a29" isRoot="false" isLeaf=" false "
    isAbstract =" false " name="Car" isSpecification="false"
    isActive =" false "></UML:Class>
  <UML:Class xmi.id="a29" isRoot="false" isLeaf=" false "
    isAbstract =" false " name="Person" isSpecification="false"
    isActive =" false "></UML:Class>
    ...
  <UML:Association xmi.id="a30" isRoot="false " isLeaf=" false "
    isAbstract =" false " name="is_manufacturer" isSpecification =" false ">
    <UML:Association.connection>
      <UML:AssociationEnd xmi.id="a31" isSpecification =" false "
        isNavigable=" false " participant ="a29">
      </UML:AssociationEnd>
      <UML:AssociationEnd xmi.id="a32"
        isSpecification =" false " isNavigable="true "
        participant ="a29">
      </UML:AssociationEnd>
    </UML:Association.connection>
  </UML:Association>
  <UML:Association xmi.id="a33" isRoot="false " isLeaf=" false "
    isAbstract =" false " name="is_owner" isSpecification =" false ">
    <UML:Association.connection>
      <UML:AssociationEnd xmi.id="a34" isSpecification =" false "
        isNavigable=" false " participant ="a29">
      </UML:AssociationEnd>
```

```
                <UML:AssociationEnd xmi.id="a35" isSpecification =" false "
                  isNavigable="true " participant ="a29">
                </UML:AssociationEnd>
              </UML:Association.connection>
            </UML:Association>
          </UML:Namespace.ownedElement>
        </UML:Package>
      </XMI.content>
    </XMI>
```

List 7-5. Sample of Translated CML in XMI without Position Information

7.4.2 Architecture Representation in UML

Architecture description could be achieved by using UML profiles. UML profile is a predefined set of extension mechanism. The UML profile defines UML as extensions as stereotypes for significant aspect in the programming language [131].

A profile is a collection of stereotype definitions, tag definitions and constraints used to represent a specific domain or target described as follows:

- Stereotypes are used to introduce a new type of model element as on extension or a classification of existing base element. (i.e. <<aspect>>) as a classification of class). The stereotype concepts allow the extended element to 'behave as if it was instantiated from the meta-model construct'.

- Tagged Values are properties for specifying characteristics or attributes for model elements. Tagged values are depicted as keyword-value pairs within property stings, written as '{tag = value}'.

- Constraints are a set of well formal rules expressed in OCL or Natural Languages. Constrains are the means by which new semantics can be introduced to UML.

Figure 7-2 shows the UML diagram for component description. The most important attributes of a component is "interface". The interface of the component is a collection of service ports provided or required that represent the direction of the connection.

Component types distinguish components by their names.

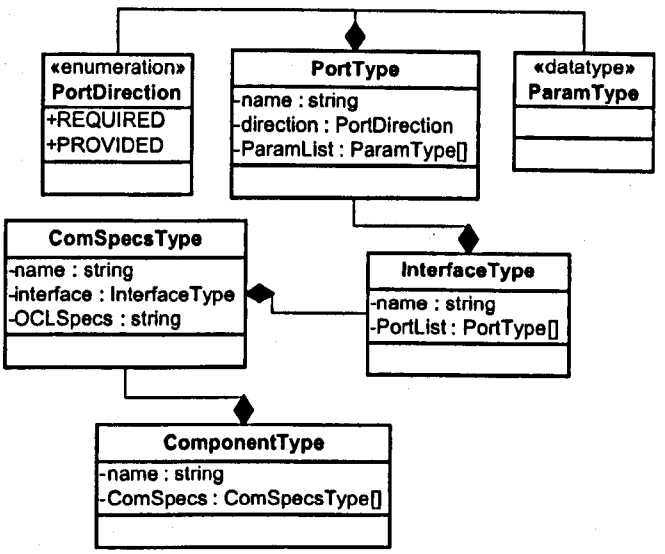


Figure 7-2. UML Diagram for Component

Figure 7-3 shows the UML diagram for connector description, consisting of 3 classes and one enumeration. The connectors have roles as counterparts of components. A connection between a component and a connector is established through the ports of the component and the corresponding roles of the connector. Predefined connector types are set by the attribute "name" of class "ConnectorType".

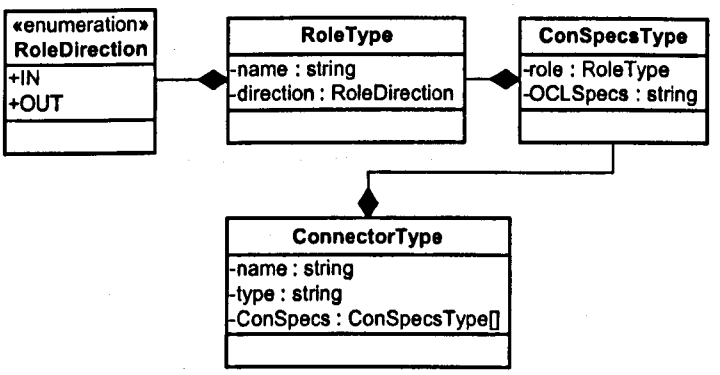


Figure 7-3. UML Diagram for Connector

Figure 7-4 shows the UML diagram for configuration, consisting of two classes. The

semantics of configuration is obtained by analysing the semantics of its connections, which in turn rely on its connectors and components.

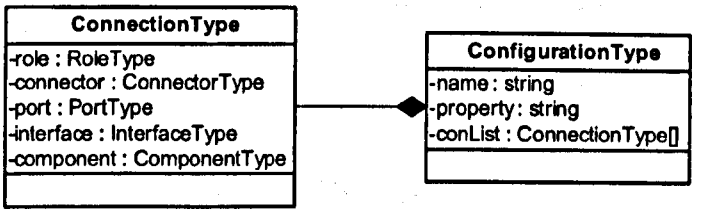


Figure 7-4. UML Diagram for Configuration

7.4.3 Feature Representation in UML

Every feature can correspond to a class. Associations between classes are tagged with a stereotype indicating the sort of feature dependency they originate from.

- The mandatory dependency is mapped to an aggregation between these classes.
- The optional dependency corresponds to an association with the cardinality of 0 or 1.
- The Oneof and Moreof lists result in abstract classes, with specific subclasses for each of the alternatives.
- The Oneof dependency results in a one-to-one association, the more-of dependency results in a one-to-many association, with multiplicity equal to the cardinality of the number of or-features.

7.5 Summary

This chapter focuses on the algorithm and implementation of model construction and model transformation based on WML and *Meta*WML.

➢ Model construction from source code of legacy system is the bridging process that transforms a program into a model. A set of model transformation rules can be used

for reverse-engineering programs into models, and analysing the properties of models.

- Model transformation in reverse engineering relies on the concept and description of abstraction, which is an effective way to reduce the complexity of software systems.

- Design patterns capture many of the structures and result from refactoring and thus provide targets for the refactorings. AOP-based model transformation is to construct weavers and apply AOP concepts by identifying possible Aspects in the underlying legacy system.

- By closely relating WML notation to UML Specification, there is assurance that this notation is UML-compatible and hence, this notation is easily convertible to the various UML exchange formats used by different UML modelling tools.

Chapter 8

Prototype Tool Support

Objectives

- To describe the architecture of prototype tool environment.
- To illustrate each tool for the proposed approach.

For modernisation of large software in line with MDA, tool support is essential. This chapter introduces a set of prototype tools, which were developed to provide help in producing models from source code and in manipulating models for abstraction and refactoring. The related modernisation tools used in REMOST approach are also discussed. All these tools were designed by the author and co-implemented by the members in the research group. Automation is a goal of tools, but with the understanding that human intervention is crucial in filling the gap of different abstraction levels, these tools can only support REMOST approach semi-automatically.

8.1 An Integration Platform

The toolset for REMOST approach provides an integration platform called FermaT Integrated Platform (FIP), which extends FermaT [166] and integrates a set of tools. In this Section, platform architecture and core functions are introduced. In the next three Sections, tools, F-ME, F-UML and F-DOC, are discussed in detail.

8.1.1 Platform Architecture

FIP is an extensible platform for software re-engineering with plug-in mechanism,

which provides a number of tools that dedicate to models and meta-models handling. Figure 8-1 shows the general system architecture of FIP, expressed in three layers: Repository, Core System and Application Plug-ins.

- Information in different transformed models and in multiple abstract views at various levels is stored in the *Repository*. A repository provides a central place to store and maintain source code and generated data.

- The *Core System* provides essential functionalities, including: (1) *Kernel Runtime*, which provides the plug-in management and communication functionalities, (2) *Transformation Engine*, which provides the program and model transformation functionalities, (3) *Visualisation Engine*, which provides easy-to-use API to create and present diagrams, and (4) *Repository Access* functionalities are used to retrieve the information from the repository.

- The *Application Plug-ins* are a set of tools, providing visualisation and analysis functionalities, for the end-users and modernisers. FIP UML (F-UML) tool, FIP Moderniser's Environment (F-ME) tool and FIP Documentation (F-DOC) tool are some examples of application plug-ins.

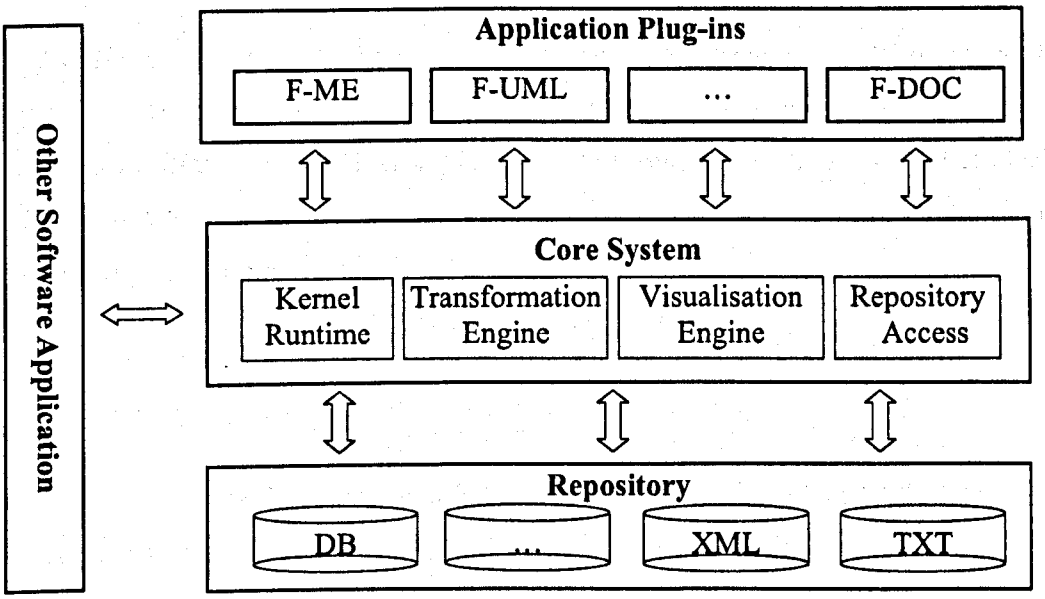


Figure 8-1. FIP Architecture

8.1.2 Platform Environment

The goal of FIP is to integrate the single tools such as F-UML, F-DOC and F-ME into one coherent toolset. To accomplish such goal, FIP Environment was developed as shown in Figure 8-2.

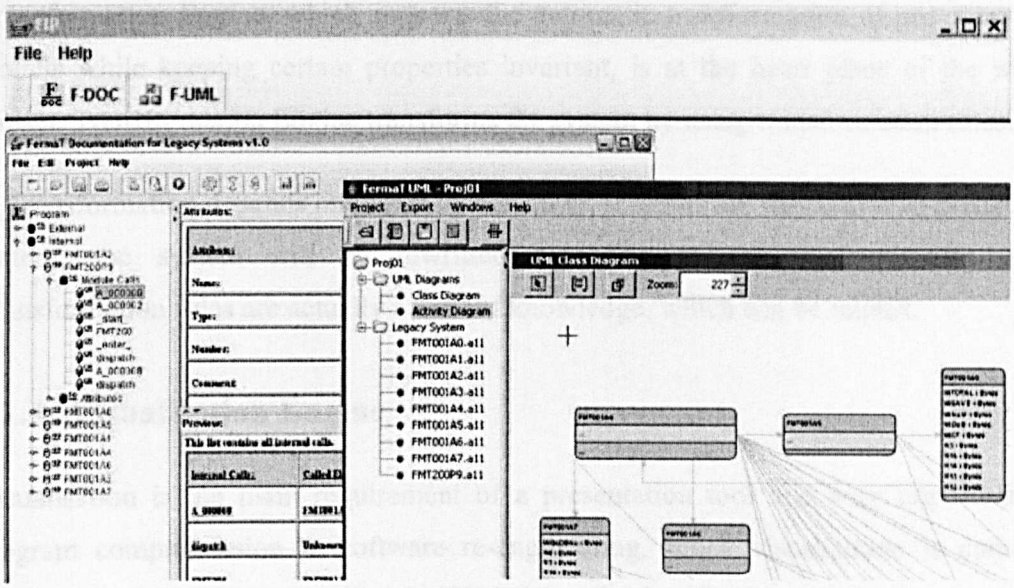


Figure 8-2. FIP Environment

FIP environment provides the plug-in mechanism that *Application Plug-ins* can be integrated into the prototype toolset. FIP environment supports multi-users in distributed environment, which is implemented through the use of Java RMI (Remote Method Invocation). FIP environment can help the modernisers go through the model driven modernisation process:

- FIP collects information from different sources and stored them in the *Repository*,
- *Transformation Engine* is used to translate the source code into WSL/WML files and transform/abstract them for migration and analysis, the new generated information is also stored in *Repository*,
- F-ME is used to manipulate WSL/WML files,

- F-UML is used to visualise the results in UML and,
- F-DOC is used to generate the documentation.

8.1.3 Transformation Engine

Transformation Engine, which supports the automatic transformation of programs and models while keeping certain properties invariant, is at the heart place of the whole toolset. Transformation Engine transforms the system by using transformation rules.

Transformation depends on matching detection. If inputs are matched with predefined pattern, the system will be rewritten according to the transformation rules. Transformation rules are actually a kind of knowledge, which can be reused.

8.1.4 Visualisation Engine

Visualisation is the main requirement of a presentation tool and very important for program comprehension in software re-engineering. Since visualisation is common functions and could be generalised, STRL Visualisation Engine (SVE) was developed as a software package, which provides an easy-to-use API to create and present the diagrams. All graphic related work could be encapsulated within the SVE so that the developers can create and browse through large scale diagrams easily without worrying about the graphical details.

The SVE consists three main parts which can be extended or modified separately:

- abstraction components to define a graph mathematically,
- graphic components for display and navigation, and
- event handling components to define the behaviour of a graph.

The concept is taken from the Model-View-Controller design pattern principle, which assures that the SVE can be extended easily and is able to handle large amounts of graphic elements.

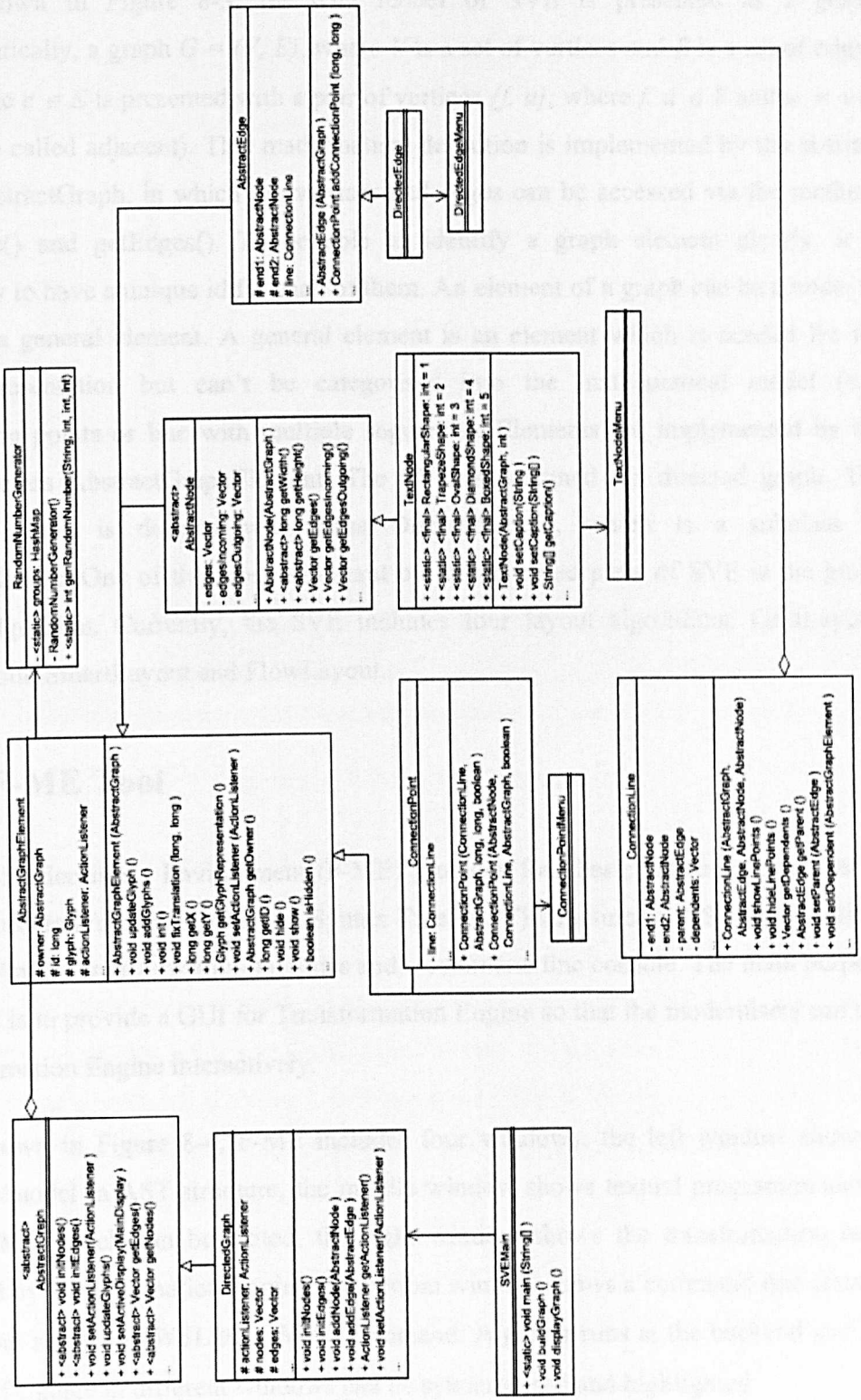


Figure 8-3. Class Diagram of SVE

As shown in Figure 8-3, the data model of SVE is presented as a graph. Mathematically, a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges. Each edge $e \in E$ is presented with a pair of vertices $\{f, u\}$, where $f, u \in V$ and $u \neq v$ (u and v are called adjacent). This mathematical definition is implemented by the abstract class AbstractGraph, in which the vertices and edges can be accessed via the methods getNodes() and getEdges(). To be able to identify a graph element clearly, it is necessary to have a unique id for each of them. An element of a graph can be a node, an edge or a general element. A general element is an element which is needed for the visual presentation but can't be categorised into the mathematical model (e.g. connection points or line with multiple segments). Elements are implemented by the abstract class AbstractGraphElement. The SVE is designed for directed graph. The directed graph is defined with class DirectedGraph, which is a subclass of AbstractGraph. One of the most important but complicated parts of SVE is the graph layout algorithm. Currently, the SVE includes four layout algorithms: GridLayout, TreeLayout, SmartLayout and FlowLayout.

8.2 F-ME Tool

FermaT Moderniser's Environment (F-ME) provides four basic functions: a parser to present program/model in Abstract Syntax Tree (AST) structure, a WSL/WML editor, program/model transformation facilities and a command line console. The main purpose of F-ME is to provide a GUI for Transformation Engine so that the modernisers can use Transformation Engine interactively.

As shown in Figure 8-4, F-ME includes four windows: the left window shows a program/model in AST structure, the middle window shows textual program/model in WSL/WML which can be edited, the right window shows the transformation rules provided by Transformation Engine, the bottom window shows a command line console which can input *Meta*WSL/*Meta*WML command. A parser runs at the backend and the results of change in different windows can be synchronised and highlighted.

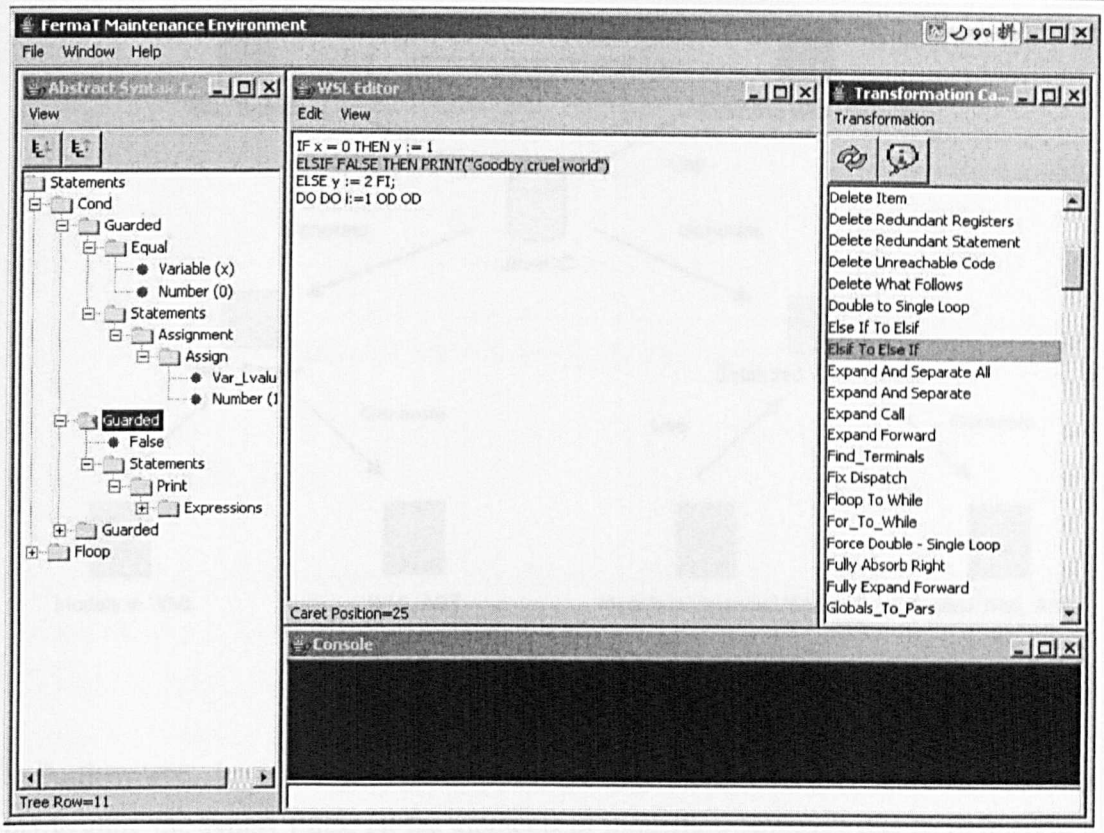


Figure 8-4. F-ME Environment

As WML is still evolving, the syntax definition for WML is not frozen, but will evolve too. Therefore, a parser that can adapter to the change of language definition is desired and Java Compiler Compiler [tm] (JavaCC [tm]) [71] is used for this purpose. JavaCC is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognise matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building, actions, debugging, etc.

With JavaCC, Language extension is easier without considering the parser implementation. Figure 8-5 shows that the language designer just needs focus on the language definition itself. The Parser and AST can be generated automatically.

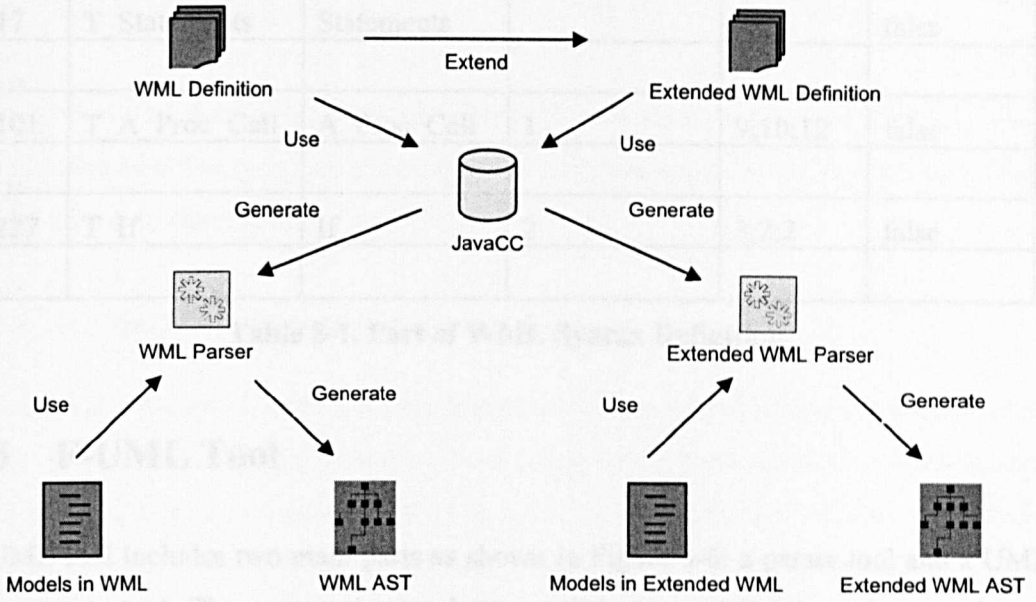


Figure 8-5. Parser Implementation

Furthermore, in order to define language that can be extended easily, without hardcoding the syntax rules, all the elements of modelling language can be defined in a table (Table 8-1 illustrates part of the WML definition).

ID	Name	Syntax Name	General Type	Children	Has Value
1	T_Statement	Statement			false
2	T_Expression	Expression			false
3	T_Condition	Condition			false
4	T_Definition	Definition			false
5	T_Lvalue	Lvalue			false
6	T_Assign	Assign		5;2	false
7	T_Guarded	Guarded		3;17	false
8	T_Action	Action		9;17	false
9	T_Name	Name			true
10	T_Expressions	Expressions		2	false
…					
12	T_Lvalues	Lvalues		5	false
…					

17	T_Statements	Statements		1	false
…					
101	T_A_Proc_Call	A_Proc_Call	1	9;10;12	false
…					
227	T_If	If	2	3;2;2	false
…					

Table 8-1. Part of WML Syntax Definition

8.3 F-UML Tool

F-UML tool includes two main parts as shown in Figure 8-6: a parser tool and a UML presentation tool. The communication between them is provided through an exchange file in XMI format. The parser tool utilises Transformation Engine to process the legacy system and extract model information, with which a MOF model is created and exported in XMI. The UML presentation tool is used to show the UML diagrams, which can be further edited. Through "cluster" and "expand/collapse" techniques, the presentation tool is also capable of handling very large diagrams.

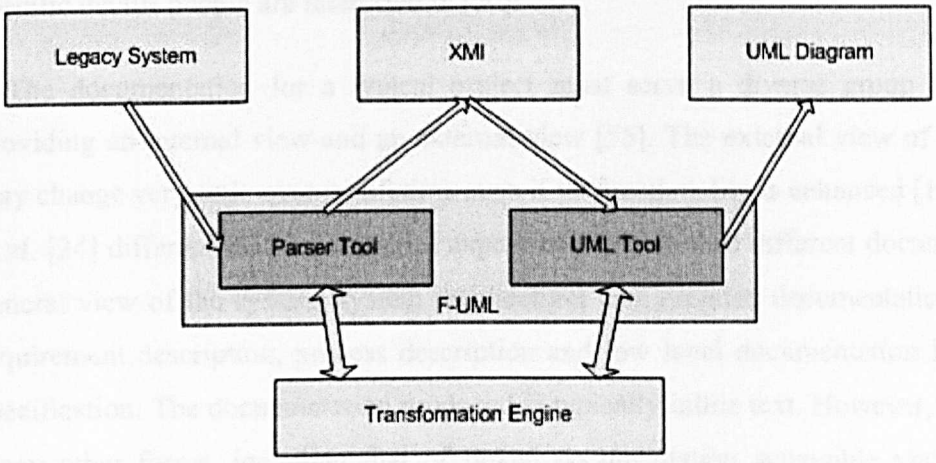


Figure 8-6. Architecture of F-UML

Currently, only two diagrams, UML class diagram and UML activity diagram, can be extracted and presented with F-UML.

8.4 F-DOC Tool

This section will discuss about the details of the re-documentation tool, F-DOC, which was developed not only as a tool, but also as a case study, presenting the rationale of the model driven modernisation.

8.4.1 Re-documentation and Environments

It is generally accepted in the software engineering that most of legacy software systems suffer from documentation problems: nonexistent or of poor quality, out-dated, over abundant and without a definite objective, difficult to access, lack of interest from the programmers, and difficult to standardise [148]. The lack of documentation is critical for software engineers and technical managers responsible for the evolution of existing software systems [155]. Without it, the only reliable and objective information is the source code itself [42]. The source code of a system can be viewed as its most detailed level of documentation: all information is there, but usually people do not have enough time to comprehend all the details. Instead, people would like to have enough information so that people can build a mental model of the system, and zoom in to the specific details people are interested in [30].

The documentation for a typical project must serve a diverse group of readers, providing an internal view and an external view [55]. The external view of a program may change very little over its lifetime even if its functionality is enhanced [155]. Cioch et al. [24] differentiate four stages of experience, which need different documentations: general view of the system; system architecture; task oriented documentations such as requirement description, process description and low level documentation like design specification. The documentation produced is typically inline text. However, it can take many other forms, including that of linked documentation accessible via hypertext, cross-reference listings, or graphical views of the software systems artefacts and relationships [155].

One way of producing accurate documentation for an existing software system is through re-documentation. Re-documentation is the creation or revision of a

semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (for example, dataflow, data structure, and control flow) intended for a human audience. Re-documentation is the simplest and oldest form of reverse engineering, and many consider it to be an un-intrusive, weak form of restructuring [22]. However, it can also be classified as a sub-area of reverse engineering because the reconstructed documentation is typically used to aid program understanding.

There already exist some tools to help re-documentation, including pretty printers (which display a code listing in an improved form), diagram generators (which create diagrams directly from code, reflecting control flow or code structure), and cross-reference listing generators. A key goal of these tools is to provide easier ways to visualise relationships among program components so people can recognise and follow paths clearly [22].

The simplest tools are to extract some documentation from the source code (e.g. javadoc, perldoc). These tools extract the signature of classes, methods, etc. and sometimes also format comments. The book format, hypertext, HTML and XML are used to improve the accessibility of documentation. The Rigi-Environment [110, 174] uses reverse engineering to reconstruct the architectural features of a legacy system, which is a typically example for structural re-documentation. The processing of a legacy system with the Rigi-Environment will be done in two steps. The first step parses the source code and stores the extracted data in a repository. The result of the first step is a resource-flow graph. This graph can be edited with a tool called Rigi-Edit which is embedded into the Rigi-Environment. The second step disassembles the resource-flow graph and analyses it to find different abstraction patterns. This step is processed semi-automatically, but needs human interaction and knowledge about the legacy system. Rajlich [133] proposes a re-documentation tool, PAS tool, which supports the incremental re-documentation. In [137], a "design browser" tool is described, for flexible browsing of a system's design level representation and for information exchange with a suite of program comprehension tools, complemented with a "retriever" supporting full-text and structural searching. Although these tool

environments can provide different views of a legacy system and are able to transform the system to another level of abstraction and generate the documentation in hypertext, no tool is model based and supports evolutionary re-documentation.

8.4.2 Model Driven Re-documentation

Traditionally, models are used as documentation. A model, produced with a modelling language, is itself a form of documentation. In fact, modelling and re-documentation both refer to the activity of describing an existing system and a quality modelling language encourages designers to write clear, self-documenting systems.

Model driven re-documentation is to produce models from existing systems that were previously produced somehow and to generate the documentation based on the models. A unique aspect of model driven re-documentation is the model based traceability. Since models can bridge the gap of a legacy system and an evolved system, the generated documentation can cover the evolutionary information of system transformation. Model driven re-documentation can provide the following benefits:

- Standardised Layouts: the documentation written in a model based manner is easy for people to read and understand it. This is also very useful when generating the template of the documentation in the hypertext style.

- Hierarchical Structure: different layer of models can produce documentations at various levels of detail. Complementing views in the model provide better description structure for understandability.

- Traceability: moving from a model to another one is usually associated to a transformation. The source of each element should be precisely identified and then the process could be reversed. Maintaining traceability links between elements of all models provide more cross reference information.

- Evolutionary: Models can bridge the gap of source systems and target systems to provide the evolutionary view of transition between legacy and evolved systems.

- Validity: Not all the models can be generated from only the source code. Human proposed models can be used to validate the machine recovered models so that the final generated documentation is consistent.

8.4.3 F-DOC Architecture and Working Process

This part follows the basic principles of Model Driven Engineering in the context of reverse engineering, discusses aspects of the re-documentation of legacy systems and proposes a model driven approach to generating documentation, which is a process of creating system documentation at different levels of abstraction and in different views of presentation. The following basic requirements have been implemented by F-DOC:

- The F-DOC creates a written documentation of the legacy system in HTML files. The documentation must be meaningful for the intended user.

- The F-DOC builds a dynamic program tree based on the MOF to navigate through the legacy system. It must also be possible to save this model as XML-file.

- The F-DOC provides a search-function which can be used to find elements in the legacy system. Elements can be not only modules, methods and attributes but also comments.

- The F-DOC provides an editor to change or extend comments in the legacy system.

- The F-DOC preserves the documentation to be consistent with the source code at all time.

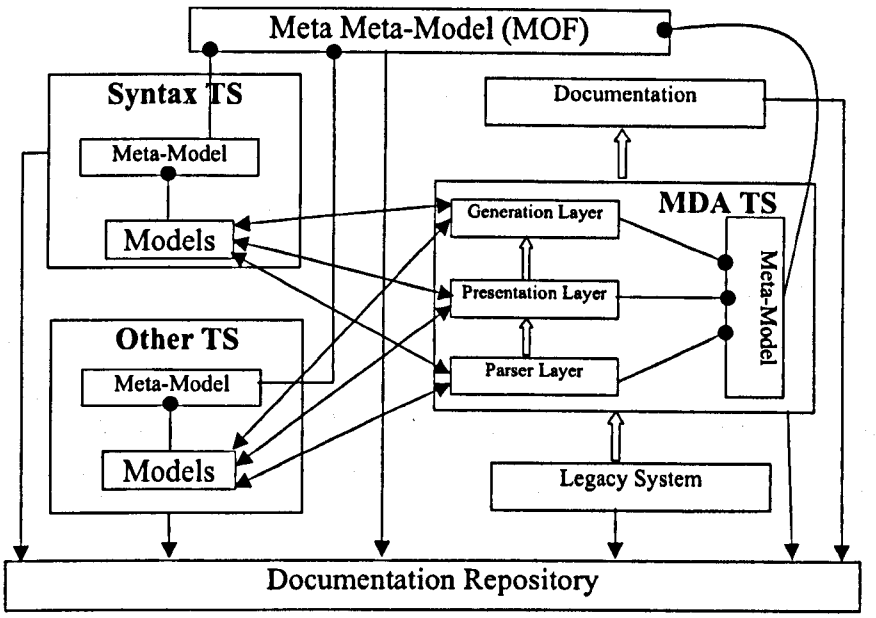


Figure 8-7. Architecture and Working Process of F-DOC

Figure 8-7 shows the architecture and working process of F-DOC. The F-DOC is designed as layer architecture, which separates the information storing, information presentation and information processing similar to Model-View-Controller (MVC) pattern.

- Documentation Repository is established to organise all information in a structured way which conforms to MDE/MDA principles. All information of source systems, target systems and transition between legacy and evolved systems can be documented so that the dependencies among program components are explicitly represented.

- The Parser Layer utilises the Transformation Engine to extract information from the repository and builds an internal image which reflects an abstract view of the legacy system. This image is organised in tree structure and can be saved as a XML file and stored in Documentation Repository.

- The Presentation Layer provides all basic program features to edit and navigate through the program tree of legacy system. The program tree is provided from the parser layer and all the changes are also saved as a XML file and stored in

Documentation Repository.

- The Creation Layer is needed to create HTML based documentation. This documentation uses hyperlinks as relationships and implements the same structure as the program tree.

Based on the above architecture, a possible process of software re-documentation is defined as follows:

- Legacy system is parsed and transformed into a stack of models. These models can be transformed into other *TSs*.
- Model information in different TSs is stored in *Documentation Repository* with a well defined meaning and can be used to produce the documentation in a uniform way.
- Information in *Documentation Repository* is presented in HTML format.

F-DOC is concerned on generated the documentation without human interaction. But practically, domain knowledge and transformation rules are the experience of experts and it is necessary to provide a user interface to allow experts to interact with the re-documentation tool by analysing and manipulating recovered design models.

8.5 Other Related Tools

This section will introduce two tools, supporting the software modernisation by using the results of model driven reverse engineering. Although these tools have not been integrated into the FIP, UML and XML compatible features make them possible to import the analysis results of existing systems into these tools for further process.

8.5.1 Aspect Oriented Weaver Tool: EvoWeaver

EvoWeaver [17] is designed for .Net platform, which aims at helping software engineers in a comprehensive process of the AOP-based software evolution. Since not

all the classes or the functions in an existing system are suitable for applying Aspect functions, five rules are proposed to analyse the existing system and help the moderniser to make a better decision.

- Rules for the Target Class Selection

Rule 1: *Aspect functions can not be applied to a class that is derived from compiled modules.*

The target class must belong to the collection of Context-bound class. The reason that ContextBoundObject is required is for clients and objects that are in the same AppDomain, which would otherwise have no proxies set up between them. Aspects are thus implemented as event sinks that get called on the message chain without any further participation or knowledge on the client's part [50].

Using this rule, the class, which is derived from a compiled class or COM object, cannot apply Aspect functions.

Rule 2: *Aspect functions can only be applied to a class that has no recursive public member functions.*

Since Delegation checks all the messages, which will be sent to the instance of target class, if there are recursive public member functions in the target class, the invocation of these functions will lead to too many checks in Delegation so that the system efficiency will be unbearable. Such a case is also mentioned in [69].

By using Rule 1 and 2, a developer can judge whether a class is suitable for applying the Aspect functions.

- Rules for Joinpoint Selection

Rule 3: *If the public member function can be invoked before the creation of the instance of target class, it cannot be defined as a Joinpoint.*

If a member function, e.g., static function, can be invoked before the creation of the instance of target class, it means that this function can be invoked before the creation of

the Delegation, and accordingly, cannot be defined as a Joinpoint.

Using this rule, a member function (e.g. static function) can be invoked before the creation of the instance of target class and cannot be defined as a Joinpoint.

- Rules for Benefits and Efficiency

Rule 4: *C_r should be at least more than 1.*

Rule 5: *If E_r is smaller than the low limitation, the system efficiency cannot be accepted.*

C_r and E_r are defined as metric functions in Section 6.2.5. If the efficiency rate cannot satisfy the requirements, Aspect functions should not be used. The low limitation is an estimative value drawn from static analysis of source code and will be different in varied applications.

By using the techniques discussed above, when a new Aspect function is needed to add into or delete from the system, the only task is to insert or delete the corresponding Aspect class node in the message chain. The Delegation makes it possible that all the functions in the system need not to be modified, which is actually a kind of transparent proxy. It means the proposed method is easy to be implemented and the structure of the evolving system will not be destroyed. It has to be mentioned that AOP technique will decrease the system efficiency.

Figure 8-8 shows the main window of EvoWeaver tool. At the top of the tool window, there are four combo-boxes, which can be used to configure an Aspect class. There are three Treeviews in Figure 8-8. Treeview 1 shows all the classes in the evolving system. By double-clicking on the node of the Treeview 1, a new window will be popped up and corresponding source code of selected class will be shown. Treeview 2 shows all the classes, which satisfy rule 1 and rule 2. These classes are shown as the tree structure according to their inherited relations. If a class in the Treeview 2 is selected, rule 3 will be applied and all the properties and selected public member functions of this class will be shown in Treeview 3. After the target classes and Aspect functions are selected, the

code conciseness rate and the efficiency rate of the evolving system can be calculated so that the developer can evaluate whether the evolved system is acceptable.

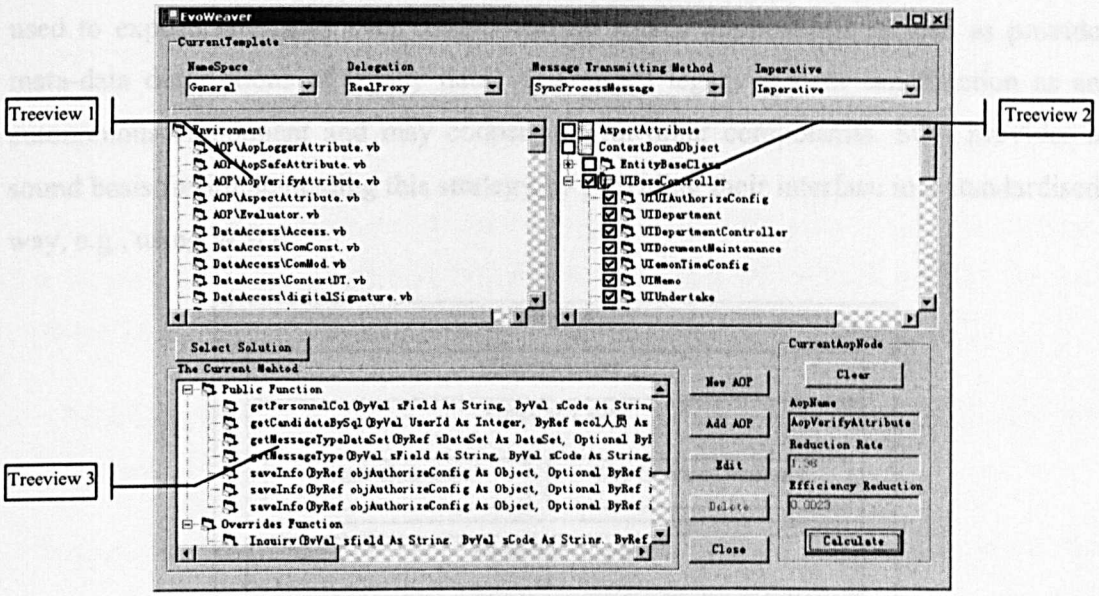


Figure 8-8. EvoWeaver Tool

List 8-1 shows the difference between the code with Aspect function and without Aspect function. The only difference is that, with AOP, <Verify()> is added before the class name. It can be done by EvoWeaver tool automatically without knowing the details of each entity class.

```
Public Class A
        Inherits EntityBaseClass
        Dim mStatisticsId As Integer
        …
End Class
        (The code without AOP)

<Verify()> _
Public Class A
        Inherits EntityBaseClass
        Dim mStatisticsId As Integer
        …
End Class
        (The code with AOP)
```

List 8-1. Difference between Code with AOP and without AOP

8.5.2 Web Services Wrapper: WSW

Web Services Wrapper (WSW) tool [57] is designed for .Net platform. A wrapper is used to expose interfaces over componentised legacy applications as well as provide meta-data descriptions of legacy data. A wrapped legacy system can function as an autonomous component and may cooperate with other components. SOA provides a sound basis for implementing this strategy by providing their interface in a standardised way, e.g., using WSDL.

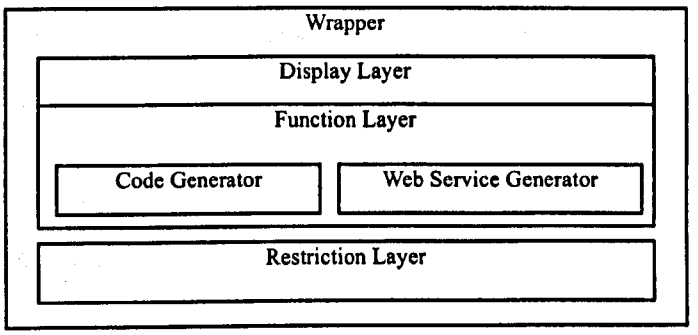


Figure 8-9. Architecture of Web Services Wrapper

Wrapping the legacy application to Web Services need not only to write plenty of wrapping code, but also need to have a strong knowledge of the original system, such as its structure, its behaviour, and its interface. Figure 8-9 shows the architecture of WSW. The wrapper is divided into three layers: Display Layer, Function Layer and Restriction Layer.

The *Display Layer* of the wrapper is mainly used to display the Web Services code and the *Wrapping Report* that is generated by the *Function Layer*. The *Function Layer* of the wrapper implements the wrapping code of Web Services. It contains *Code Generator* and *Web Services Generator*. *Code Generator* generates the Web Services implementation code according to the properties of Web Services and Web Services method, which are set by the developers. *Web Services Generator* calls the compiler of Microsoft .Net to compile the Web Services implementation code and then deploys the Web Services. *Restriction Layer* declares the restrictions of wrapping. Before generating the Web Services, the *Web Services Generator* will validate whether the

wrapping methods satisfy these restrictions. Currently, four restrictions are defined:

- **Restriction 1:** *the type of the method must be public.*

  Only public methods are useful to be implemented as Web Services methods.

- **Restriction 2:** *abstract methods cannot be wrapped into Web Services method.*

The access of the Web Services method must arouse the execution of the method. The abstract methods, which only define the framework of methods without the real business logic, are not able to be implemented as Web Services methods.

- **Restriction 3:** *overload methods must have different names of Web Services methods.*

Overload methods mean that two or more methods have the same method name, but the parameters are different. If the overload methods should be wrapped into Web Services, they should have different names.

- **Restriction 4:** *if the original method contains transaction and it is not the root object of the transaction, the method cannot be wrapped into the Web Services method.*

The goal of transaction is to maintain the data integrity. All the update operations should be success entirely, or else should be failure entirely. Only the methods which initiate the transaction should be wrapped into Web Services.

Figure 8-10 shows the main form of WSW. On the basis of the analysis of the legacy application, developers decide which classes can be wrapped into Web Services and which methods can be wrapped into Web Services method according to the needs of integration. Before WSW generates the Web Services and related wrapping code, developers should set the properties of Web Services and Web Services method. The properties of Web Services are service name, description and namespace. The properties of Web Services method are MessageName, CacheDuration, EnableSession, TransactionOption, BufferResponse, and Description etc. With these property settings,

WSW can further check whether the selected methods satisfy the wrapping restrictions. Validated Web Services and Web Services methods can be wrapped automatically. List 8-2 shows a slice of generated Web Services code.

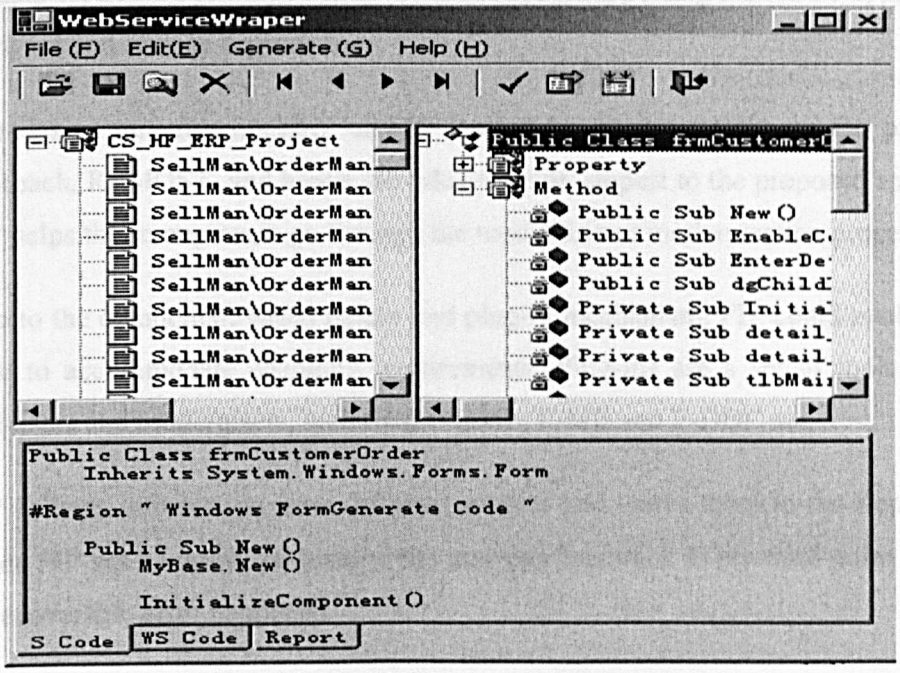


Figure 8-10. Main Form of WSW

```
Imports System.Web.Services
<System.Web.Services.WebService( Description:="SaleOrderService",
  Name:="SaleBill", NameSpace:="URL of the Company")>
  Public Class SaleBill
    InHerits System.Web.Services.WebService
      Public Sub New()
        MyBase.New()
      End Sub
<WebMethod(
  BufferResponse:=True, CacheDuration:="0", Description:="QuerySaleBill",
  MessageName:="QueryBySql",TransactionOption:="Disabled">
  Public Function Query(ByVal sSql As String, ByRef dt As DataSet,
                        Optional ByRef sErrDescr As String="") As Boolean
    ...
End Function
```

List 8-2. Generated Web Services

8.6 Summary

In this chapter, a set of prototype tools are introduced, supporting REMOST approach to both forward and reverse engineering.

- The design and functionality of FIP toolset are based exactly on the proposed approach, REMOST, and hence provide coherent support to the proposed approach. FIP helps the re-engineers go through the model driven modernisation process

- Due to the component-based nature and plug-in mechanism, FIP could evolve over time to accommodate changing requirements. Plug-ins are a set of tools for the end-users and modernisers, including F-UML, F-ME and F-DOC etc.

- FIP collects information from different sources and stores them in the Repository. Since FIP has an operation history, the user can backtrack to previous point once an unrecoverable error happened.

- Transformation Engine is used to translate the source code into WSL/WML files and transform/abstract them for migration and analysis. All the transformations and abstractions rules can be done automatically.

- SVE was developed for visualisation, which provides an easy-to-use API to create and present the diagrams. All graphic related work could be encapsulated within the engine so that the developers can easily create and browse through large scale diagrams without worrying about the graphical details.

- The WSL/WML files can be manipulated by F-ME, which provides four basic functions: a parser to present program/model in AST structure, a WSL/WML editor, transformation manipulation and a command line console. As WML is still evolving, the syntax definition for WML is not frozen, but will evolve too. Therefore, a parser that can adapter to the change of language definition is desired and JavaCC is used for this purpose.

- F-UML includes two main parts: a parser tool and an UML presentation tool. The

parser tool utilises Transformation Engine to process the legacy system and extract model information, The UML presentation tool is used to show and edit the UML diagrams. The communication between them is provided through an exchange file in XMI format.

- F-DOC was developed as a tool for the re-documentation, but also as a case study, presenting the rationale of the model driven reverse engineering.
- EvoWeaver is designed for .Net platform, which aims at helping software engineers in a comprehensive process of the AOP-based software evolution.
- Web Services Wrapper (WSW) is used to wrap the legacy application to Web Services by exposing in a standardised way.

Chapter 9
Case Study

Objectives

- To show how to use REMOST to develop a modernisation strategy for different sorts of legacy systems.
- To show how to combine different modelling techniques for software modernisation.
- To show how to use FIP and other tools in software modernisation process.

9.1 Overview

Software modernisation normally involves many developers with diverse backgrounds, large systems with many sub-systems, and serials of change requirements. A number of factors affect the progress and results of modernisation tasks. As a consequence, the case studies have been chosen as the validation technique because it is the research method, which provide a systematic way of looking at events, collecting data, analysing information, and reporting the results. Case studies lend themselves to both generating and testing hypotheses [43].

The theme of this thesis can be decomposed into three claims. A first claim is that REMOST approach can help modernisers perform software modernisation tasks more systematically. This claim will be referred to as the usefulness claim. A second claim is that REMOST approach can be produced cost-effectively during whole software modernisation. This claim will be referred to as the effectiveness claim. Finally, the

third claim is that REMOST can be used to support software modernisation on different kinds and scales of system. This will be referred to as the completeness claim. To validate these claims, three selected case studies using REMOST approach and FIP toolset have been experienced. Each case study was focusing on different claims above. The completeness is claim by three case studies themselves since they are very different in size, platform and application area.

The first case study is an Assembler Migration (AM) example, which helps to illustrate detailed steps for using REMOST method. This case study focuses on evaluating the usefulness claim and provides guidance to readers to use the FIP tool in their own practice.

In the second case study, REMOST is used to investigate the Platform Migration (PM) from one Real Time Operation System (RTOS) to other RTOS. The PM case study focuses on validating the low cost claim that software could be migrated on a new platform effectively.

The third case study (AgenEvo) illustrates how to modernise legacy system into agent-based web services.

Case Study	System size	Usefulness	Effectiveness	Completeness
AM	1,000LOC	√		√
PM	4,000LOC		√	√
AgenEvo	500LOC	√		√

Table 9-1. Properties of Three Case Studies

Case Study	FDL	ADL	CML	WSL	UML
AM			√	√	√
PM	√		√	√	
AgenEvo		√	√	√	

Table 9-2. Modelling Languages Used in Three Case Studies

Table 9-1 summarises the claim each study focused on, and the characteristics of each study. The first two columns list the name of the system evolved as part of each study and its size in lines of code. Table 9-2 summarises the modelling languages used in each case study. In the rest of this chapter, each case study follows a prescribed format: Background (the purpose, a description of the system under study), Process (the language used, the process used, the results), and Discussion (an evaluation).

9.2 Assembler Migration (AM)

9.2.1 Background

The case study performed for this research tried from a small assembly language project with 11 files, which is a package of bank application. Model-driven development claims to offer the same improvements to developers that the adoption of procedural languages from assembly language [81, 100]. The purpose of this study is to see if FIP transformations can be used to cover the whole re-engineering process from very low-level code (assembler) to high level model specifications.

The case study starts with an IBM 370 assembler module that is translated to WSL and re-engineered to an abstract specification in WML. The FIP tool collects information from different sources and uses the Transformation Engine to translate the assembler files into WSL files. Once this conversion to WSL has been completed, WSL files are transformed/abstracted into WML and further extracted into UML models. These experiments have shown that programs that have been transformed using the tool can be expressed in a higher level abstraction form that subjectively is much easier to understand than the original. Figure 9-1 shows an example of different models and views for a slice of code in assembler.

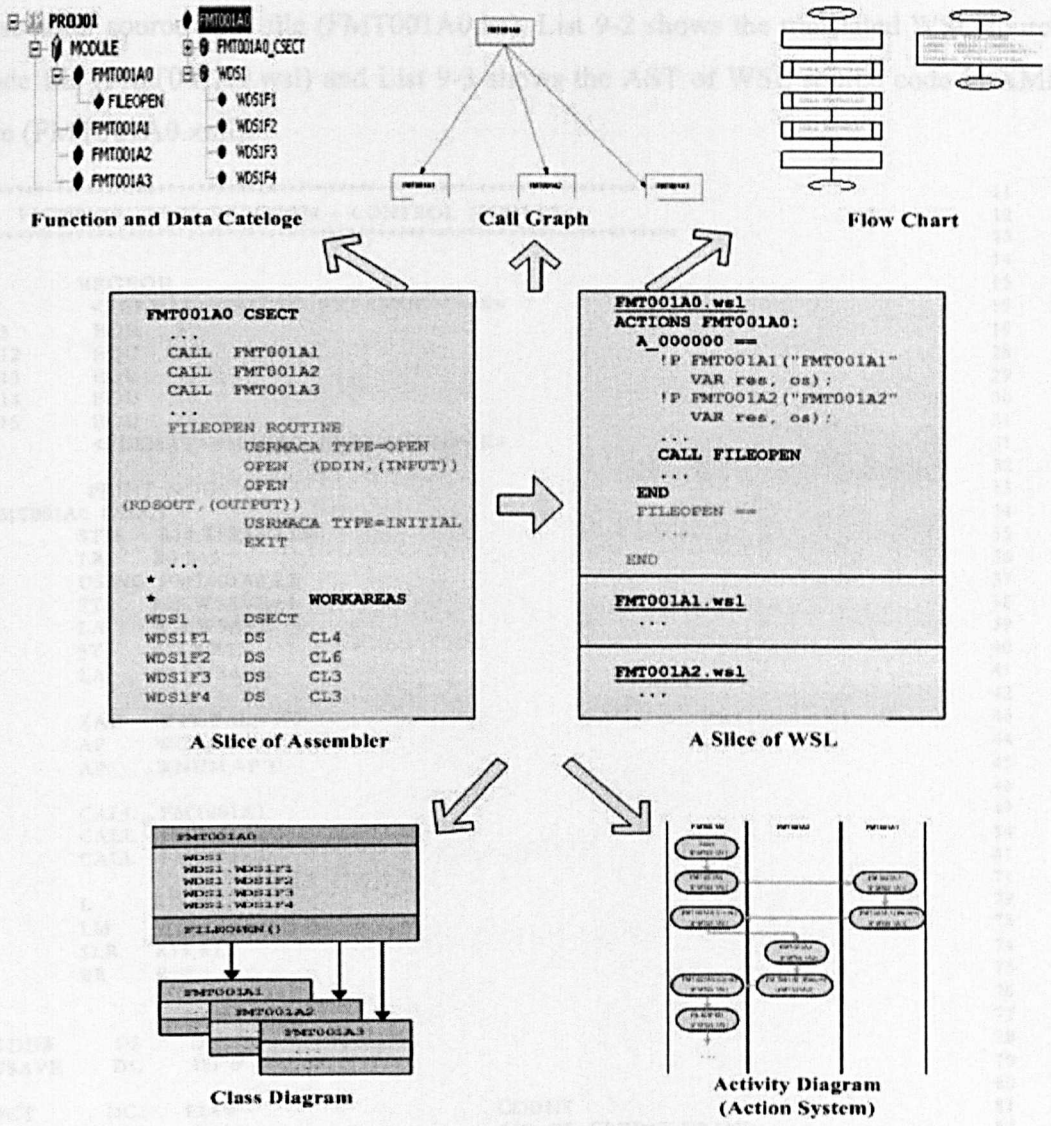


Figure 9-1. Different Models and Views for a Slice of Code

9.2.2 Assembler to WSL translation and WSL Transformation

The aim of the assembler-to-WSL translator is to generate WSL code that models as accurately as possible the behavior of the original assembler module, without worrying too much about the size, efficiency, or complexity of the resulting code. This step can be done by FermaT transformation engine [168, 180]. The FermaT transformation engine includes some very powerful transformations for such tasks as simplifying WSL code, removing redundancies, and tracking dispatch codes. List 9-1 shows the

assembler source code file (FMT001A0.lst), List 9-2 shows the translated WSL source code file (FMT001A0.wsl) and List 9-3 shows the AST of WSL source code in XML file (FMT001A0.xml).

```
*****************************************************************                11
*  FMT001A0  TEST PROGRAM - CONTROL MODULE                      *                12
*****************************************************************                13
*                                                                                14
         REGEQU                                                                  15
*         <FERMAT><MACRO EXPANSION><S>                                          19
R3        EQU   3                                                               19
R12       EQU   12                                                              28
R13       EQU   13                                                              29
R14       EQU   14                                                              30
R15       EQU   15                                                              31
*         <FERMAT><MACRO EXPANSION><E>                                          31
*                                                                               32
*         PRINT NOGEN                                                           33
FMT001A0 CSECT                                                                  34
         STM   R14,R12,12(R13)                                                  35
         LR    R3,R15                                                           36
         USING FMT001A0,R3                                                      37
         ST    R13,WSAVE+4                                                      38
         LA    R14,WSAVE                                                        39
         ST    R14,8(R13)                                                       40
         LA    R13,WSAVE                                                        41
*                                                                               42
         ZAP   WTOTAL,=P'0'                                                     43
         AP    WCT,=P'1'                                                        44
         AP    WNUM,=P'1'                                                       45
*                                                                               46
         CALL  FMT001A1                                                         47
         CALL  FMT001A2                                                         54
         CALL  FMT001A3                                                         61
*                                                                               71
         L     R13,WSAVE+4                                                      72
         LM    R14,R12,12(R13)                                                  73
         SLR   R15,R15                                                          74
         BR    R14                                                              75
*                                                                               76
*              WORKAREAS                                                        77
WDUB     DS    D                                                                78
WSAVE    DC    18F'0'                                                           79
*                                                                               80
WCT      DC    PL4'0'                  COUNT                                    81
WNUM     DC    PL4'0'                  NO OF CREDIT TRANS                       82
WTOTAL   DC    PL4'0'                   AMOUNT                                  83
                                                                                84
         LTORG                                                                  85
*                                                                               88
         END                                                                    89
```

List 9-1. Assembler Source Code of Module FMT001A0

```
VAR < cc := 0, cc1 := 0, destination := 0 >:
ACTIONS _enter_:
  _enter_ ==
    C:" <ENTRY POINT> ";
    C:" <NAME=FMT001A0> ";
    r7 := __r7_init__;
    r11 := __r11_init__;
    r12 := __r12_init__;
    r13 := __r13_init__;
    r14 := __r14_init__;
    CALL FMT001A0 END
```

```
_start_ == CALL Z END
FMT001A0 ==
   C:"*************************************************************************";
   C:"*FMT001A0 TEST PROGRAM - CONTROL MODULE                                 *";
   C:"*************************************************************************";
   C:"*";
   C:"*";
   C:"*            PRINT NOGEN";
   CALL A_000000 END
A_000000 ==
   C:"<FermaT> 00000035 ";
   !P push_regs(r0,
                        r1,
                        r2,
                        r3,
                        r4,
                        r5,
                        r6,
                        r7,
                        r8,
                        r9,
                        r10,
                        r11,
                        r12,
                        r13,
                        r14
      VAR reg_stack);
   CALL A_000004 END
A_000004 ==
   C:"<FermaT> 00000036 "; r3 := r15; CALL A_000006
END
A_000006 ==
   C:"<FermaT> 00000038 WSAVE WSAVE[5] WSAVE[6] WSAVE[7] WSAVE[8]";
   WSAVE[5..8] := r13;
   CALL A_00000A END
A_00000A ==
   C:"<FermaT> 00000039 WSAVE";
   r14 := !XF address_of(WSAVE);
   CALL A_00000E END
A_00000E ==
   C:"<FermaT> 00000040 ";
   a[r13 + 8, 4] := r14;
   CALL A_000012 END
A_000012 ==
   C:"<FermaT> 00000041 WSAVE";
   r13 := !XF address_of(WSAVE);
   CALL A_000016 END
A_000016 ==
   C:"*";
   C:"<FermaT> 00000043 WTOTAL";
   !P zap(!XF p_lit(1, 1, "0") VAR WTOTAL);
   IF !XC dec_eq(WTOTAL, !XF p_lit(1, 4, "0"))
      THEN cc := 0
   ELSIF !XC dec_less(WTOTAL, !XF p_lit(1, 4, "0"))
      THEN cc := 1
      ELSE cc := 2 FI;
   CALL A_00001C END
A_00001C ==
```

```
    C:"<FermaT> 00000044 WCT";
    !P ap(!XF p_lit(1, 1, "1") VAR WCT);
    IF !XC dec_eq(WCT, !XF p_lit(1, 4, "0"))
      THEN cc := 0
    ELSIF !XC dec_less(WCT, !XF p_lit(1, 4, "0"))
      THEN cc := 1
      ELSE cc := 2 FI;
    CALL A_000022 END
A_000022 ==
    C:"<FermaT> 00000045 WNUM";
    !P ap(!XF p_lit(1, 1, "1") VAR WNUM);
    IF !XC dec_eq(WNUM, !XF p_lit(1, 4, "0"))
      THEN cc := 0
    ELSIF !XC dec_less(WNUM, !XF p_lit(1, 4, "0"))
      THEN cc := 1
      ELSE cc := 2 FI;
    CALL A_000028 END
A_000028 ==
    C:"*";
    C:"<FermaT> 00000047 ";
    !P FMT001A1("FMT001A1" VAR result_code, os);
    r15 := result_code;
    CALL A_000036 END
A_000036 ==
    C:"<FermaT> 00000054 ";
    !P FMT001A2("FMT001A2" VAR result_code, os);
    r15 := result_code;
    CALL A_000046 END
A_000046 ==
    C:"<FermaT> 00000061 ";
    !P FMT001A3("FMT001A3" VAR result_code, os);
    r15 := result_code;
    CALL A_000056 END
A_000056 ==
    C:"*";
    C:"<FermaT> 00000072 WSAVE WSAVE[5] WSAVE[6] WSAVE[7] WSAVE[8]";
    r13 := WSAVE[5..8];
    CALL A_00005A END
A_00005A ==
    C:"<FermaT> 00000073 ";
    !P pop_regs(
      VAR r0, r1,
        r2,
        r3,
        r4,
        r5,
        r6,
        r7,
        r8,
        r9,
        r10,
        r11,
        r12,
        r13,
        r14,
        reg_stack);
    CALL A_00005E END
A_00005E ==
```

```
    C:"<FermaT> 00000074 ";
    r15 := 0;
    IF r15 = 0 THEN cc := 0 ELSE cc := 2 FI;
    CALL A_000060 END
  A_000060 ==
    C:"<FermaT> 00000075 ";
    destination := r14;
    CALL dispatch END
  dispatch ==
    IF destination = 0
      THEN CALL Z
      ELSE C:" Unknown destination "; CALL Z FI END
ENDACTIONS ENDVAR
```

List 9-2. WSL Source Code of Module FMT001A0

```
<Statements>
  <Var>
    <Assigns>
      <Assign>
        <Var_Lvalue value="cc"></Var_Lvalue>
        <Number value="0"></Number>
      </Assign>
      <Assign>
        <Var_Lvalue value="cc1"></Var_Lvalue>
        <Number value="0"></Number>
      </Assign>
      <Assign>
        <Var_Lvalue value="destination"></Var_Lvalue>
        <Number value="0"></Number>
      </Assign>
    </Assigns>
    <Statements>
      <A_S>
        <Name value="_enter_"></Name>
        <Actions>
          <Action>
            <Name value="_enter_"></Name>
            <Statements>
              <Comment value="&lt;ENTRY POINT&gt; "></Comment>
              <Comment value="&lt;NAME=FMT001A0&gt; "></Comment>
              <Assignment>
                <Assign>
                  <Var_Lvalue value="r7"></Var_Lvalue>
                  <Variable value="__r7_init__"></Variable>
                </Assign>
              </Assignment>
              <Assignment>
                <Assign>
                  <Var_Lvalue value="r11"></Var_Lvalue>
                  <Variable value="__r11_init__"></Variable>
                </Assign>
              </Assignment>
              <Assignment>
                <Assign>
                  <Var_Lvalue value="r12"></Var_Lvalue>
                  <Variable value="__r12_init__"></Variable>
```

```
                    </Assign>
                </Assignment>
                <Assignment>
                    <Assign>
                        <Var_Lvalue value="r13"></Var_Lvalue>
                        <Variable value="__r13_init__"></Variable>
                    </Assign>
                </Assignment>
                <Assignment>
                    <Assign>
                        <Var_Lvalue value="r14"></Var_Lvalue>
                        <Variable value="__r14_init__"></Variable>
                    </Assign>
                </Assignment>
                <Call value="FMT001A0"></Call>
            </Statements>
        </Action>
        <Action>
            <Name value="_start_"></Name>
            <Statements>
                <Call value="Z"></Call>
            </Statements>
        </Action>
        <Action>
            <Name value="FMT001A0"></Name>
            <Statements>
                <Comment
value="*****************************************************************************"></Comment>
                <Comment value="*   FMT001A0   TEST PROGRAM - CONTROL
MODULE                              *"></Comment>
                <Comment
value="*****************************************************************************"></Comment>
                <Comment value="*"></Comment>
                <Comment value="*"></Comment>
                <Comment value="*            PRINT NOGEN"></Comment>
                <Call value="A_000000"></Call>
            </Statements>
        </Action>
        <Action>
            <Name value="A_000000"></Name>
            <Statements>
                <Comment value="&amp;lt;FermaT&amp;gt; 00000035 "></Comment>
                <A_Proc_Call>
                    <Name value="push_regs"></Name>
                    <Expressions>
                        <Variable value="r0"></Variable>
                        <Variable value="r1"></Variable>
                        <Variable value="r2"></Variable>
                        <Variable value="r3"></Variable>
                        <Variable value="r4"></Variable>
                        <Variable value="r5"></Variable>
                        <Variable value="r6"></Variable>
                        <Variable value="r7"></Variable>
                        <Variable value="r8"></Variable>
                        <Variable value="r9"></Variable>
                        <Variable value="r10"></Variable>
                        <Variable value="r11"></Variable>
                        <Variable value="r12"></Variable>
```

```
                    <Variable value="r13"></Variable>
                    <Variable value="r14"></Variable>
                </Expressions>
                <Lvalues>
                    <Var_Lvalue value="reg_stack"></Var_Lvalue>
                </Lvalues>
            </A_Proc_Call>
            <Call value="A_000004"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_000004"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000036 "></Comment>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r3"></Var_Lvalue>
                    <Variable value="r15"></Variable>
                </Assign>
            </Assignment>
            <Call value="A_000006"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_000006"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000038 WSAVE WSAVE[5]
              WSAVE[6] WSAVE[7] WSAVE[8]"></Comment>
            <Assignment>
                <Assign>
                    <Sub_Seg_Lvalue>
                        <Var_Lvalue value="WSAVE"></Var_Lvalue>
                        <Number value="5"></Number>
                        <Number value="8"></Number>
                    </Sub_Seg_Lvalue>
                    <Variable value="r13"></Variable>
                </Assign>
            </Assignment>
            <Call value="A_00000A"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_00000A"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000039 WSAVE"></Comment>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r14"></Var_Lvalue>
                    <X_Funct_Call>
                        <Name value="address_of"></Name>
                        <Expressions>
                            <Variable value="WSAVE"></Variable>
                        </Expressions>
                    </X_Funct_Call>
                </Assign>
            </Assignment>
            <Call value="A_00000E"></Call>
        </Statements>
```

```
</Action>
<Action>
    <Name value="A_00000E"></Name>
    <Statements>
        <Comment value="&lt;FermaT&gt; 00000040 "></Comment>
        <Assignment>
            <Assign>
                <Rel_Seg_Lvalue>
                    <Var_Lvalue value="a"></Var_Lvalue>
                    <Plus>
                        <Variable value="r13"></Variable>
                        <Number value="8"></Number>
                    </Plus>
                    <Number value="4"></Number>
                </Rel_Seg_Lvalue>
                <Variable value="r14"></Variable>
            </Assign>
        </Assignment>
        <Call value="A_000012"></Call>
    </Statements>
</Action>
<Action>
    <Name value="A_000012"></Name>
    <Statements>
        <Comment value="&lt;FermaT&gt; 00000041 WSAVE"></Comment>
        <Assignment>
            <Assign>
                <Var_Lvalue value="r13"></Var_Lvalue>
                <X_Funct_Call>
                    <Name value="address_of"></Name>
                    <Expressions>
                        <Variable value="WSAVE"></Variable>
                    </Expressions>
                </X_Funct_Call>
            </Assign>
        </Assignment>
        <Call value="A_000016"></Call>
    </Statements>
</Action>
<Action>
    <Name value="A_000016"></Name>
    <Statements>
        <Comment value="*"></Comment>
        <Comment value="&lt;FermaT&gt; 00000043 WTOTAL"></Comment>
        <A_Proc_Call>
            <Name value="zap"></Name>
            <Expressions>
                <X_Funct_Call>
                    <Name value="p_lit"></Name>
                    <Expressions>
                        <Number value="1"></Number>
                        <Number value="1"></Number>
                        <String value="0"></String>
                    </Expressions>
                </X_Funct_Call>
            </Expressions>
            <Lvalues>
                <Var_Lvalue value="WTOTAL"></Var_Lvalue>
```

```
            </Lvalues>
        </A_Proc_Call>
        <Cond>
            <Guarded>
                <X_BFunct_Call>
                    <Name value="dec_eq"></Name>
                    <Expressions>
                        <Variable value="WTOTAL"></Variable>
                        <X_Funct_Call>
                            <Name value="p_lit"></Name>
                            <Expressions>
                                <Number value="1"></Number>
                                <Number value="4"></Number>
                                <String value="0"></String>
                            </Expressions>
                        </X_Funct_Call>
                    </Expressions>
                </X_BFunct_Call>
                <Statements>
                    <Assignment>
                        <Assign>
                            <Var_Lvalue value="cc"></Var_Lvalue>
                            <Number value="0"></Number>
                        </Assign>
                    </Assignment>
                </Statements>
            </Guarded>
            <Guarded>
                <X_BFunct_Call>
                    <Name value="dec_less"></Name>
                    <Expressions>
                        <Variable value="WTOTAL"></Variable>
                        <X_Funct_Call>
                            <Name value="p_lit"></Name>
                            <Expressions>
                                <Number value="1"></Number>
                                <Number value="4"></Number>
                                <String value="0"></String>
                            </Expressions>
                        </X_Funct_Call>
                    </Expressions>
                </X_BFunct_Call>
                <Statements>
                    <Assignment>
                        <Assign>
                            <Var_Lvalue value="cc"></Var_Lvalue>
                            <Number value="1"></Number>
                        </Assign>
                    </Assignment>
                </Statements>
            </Guarded>
            <Guarded>
                <True></True>
                <Statements>
                    <Assignment>
                        <Assign>
                            <Var_Lvalue value="cc"></Var_Lvalue>
                            <Number value="2"></Number>
```

```
                    </Assign>
                  </Assignment>
                </Statements>
              </Guarded>
            </Cond>
            <Call value="A_00001C"></Call>
          </Statements>
        </Action>
        <Action>
          <Name value="A_00001C"></Name>
          <Statements>
            <Comment value="&amp;lt;FermaT&amp;gt; 00000044 WCT"></Comment>
            <A_Proc_Call>
              <Name value="ap"></Name>
              <Expressions>
                <X_Funct_Call>
                  <Name value="p_lit"></Name>
                  <Expressions>
                    <Number value="1"></Number>
                    <Number value="1"></Number>
                    <String value="1"></String>
                  </Expressions>
                </X_Funct_Call>
              </Expressions>
              <Lvalues>
                <Var_Lvalue value="WCT"></Var_Lvalue>
              </Lvalues>
            </A_Proc_Call>
            <Cond>
              <Guarded>
                <X_BFunct_Call>
                  <Name value="dec_eq"></Name>
                  <Expressions>
                    <Variable value="WCT"></Variable>
                    <X_Funct_Call>
                      <Name value="p_lit"></Name>
                      <Expressions>
                        <Number value="1"></Number>
                        <Number value="4"></Number>
                        <String value="0"></String>
                      </Expressions>
                    </X_Funct_Call>
                  </Expressions>
                </X_BFunct_Call>
                <Statements>
                  <Assignment>
                    <Assign>
                      <Var_Lvalue value="cc"></Var_Lvalue>
                      <Number value="0"></Number>
                    </Assign>
                  </Assignment>
                </Statements>
              </Guarded>
              <Guarded>
                <X_BFunct_Call>
                  <Name value="dec_less"></Name>
                  <Expressions>
                    <Variable value="WCT"></Variable>
```

```
                                <X_Funct_Call>
                                    <Name value="p_lit"></Name>
                                    <Expressions>
                                        <Number value="1"></Number>
                                        <Number value="4"></Number>
                                        <String value="0"></String>
                                    </Expressions>
                                </X_Funct_Call>
                            </Expressions>
                        </X_BFunct_Call>
                        <Statements>
                            <Assignment>
                                <Assign>
                                    <Var_Lvalue value="cc"></Var_Lvalue>
                                    <Number value="1"></Number>
                                </Assign>
                            </Assignment>
                        </Statements>
                    </Guarded>
                    <Guarded>
                        <True></True>
                        <Statements>
                            <Assignment>
                                <Assign>
                                    <Var_Lvalue value="cc"></Var_Lvalue>
                                    <Number value="2"></Number>
                                </Assign>
                            </Assignment>
                        </Statements>
                    </Guarded>
                </Cond>
                <Call value="A_000022"></Call>
            </Statements>
        </Action>
        <Action>
            <Name value="A_000022"></Name>
            <Statements>
                <Comment value="&amp;lt;FermaT&amp;gt; 00000045 WNUM"></Comment>
                <A_Proc_Call>
                    <Name value="ap"></Name>
                    <Expressions>
                        <X_Funct_Call>
                            <Name value="p_lit"></Name>
                            <Expressions>
                                <Number value="1"></Number>
                                <Number value="1"></Number>
                                <String value="1"></String>
                            </Expressions>
                        </X_Funct_Call>
                    </Expressions>
                    <Lvalues>
                        <Var_Lvalue value="WNUM"></Var_Lvalue>
                    </Lvalues>
                </A_Proc_Call>
                <Cond>
                    <Guarded>
                        <X_BFunct_Call>
                            <Name value="dec_eq"></Name>
```

```
                    <Expressions>
                        <Variable value="WNUM"></Variable>
                        <X_Funct_Call>
                            <Name value="p_lit"></Name>
                            <Expressions>
                                <Number value="1"></Number>
                                <Number value="4"></Number>
                                <String value="0"></String>
                            </Expressions>
                        </X_Funct_Call>
                    </Expressions>
                </X_BFunct_Call>
                <Statements>
                    <Assignment>
                        <Assign>
                            <Var_Lvalue value="cc"></Var_Lvalue>
                            <Number value="0"></Number>
                        </Assign>
                    </Assignment>
                </Statements>
            </Guarded>
            <Guarded>
                <X_BFunct_Call>
                    <Name value="dec_less"></Name>
                    <Expressions>
                        <Variable value="WNUM"></Variable>
                        <X_Funct_Call>
                            <Name value="p_lit"></Name>
                            <Expressions>
                                <Number value="1"></Number>
                                <Number value="4"></Number>
                                <String value="0"></String>
                            </Expressions>
                        </X_Funct_Call>
                    </Expressions>
                </X_BFunct_Call>
                <Statements>
                    <Assignment>
                        <Assign>
                            <Var_Lvalue value="cc"></Var_Lvalue>
                            <Number value="1"></Number>
                        </Assign>
                    </Assignment>
                </Statements>
            </Guarded>
            <Guarded>
                <True></True>
                <Statements>
                    <Assignment>
                        <Assign>
                            <Var_Lvalue value="cc"></Var_Lvalue>
                            <Number value="2"></Number>
                        </Assign>
                    </Assignment>
                </Statements>
            </Guarded>
        </Cond>
        <Call value="A_000028"></Call>
```

```
        </Statements>
    </Action>
    <Action>
        <Name value="A_000028"></Name>
        <Statements>
            <Comment value="*"></Comment>
            <Comment value="&lt;FermaT&gt; 00000047 "></Comment>
            <A_Proc_Call>
                <Name value="FMT001A1"></Name>
                <Expressions>
                    <String value="FMT001A1"></String>
                </Expressions>
                <Lvalues>
                    <Var_Lvalue value="result_code"></Var_Lvalue>
                    <Var_Lvalue value="os"></Var_Lvalue>
                </Lvalues>
            </A_Proc_Call>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r15"></Var_Lvalue>
                    <Variable value="result_code"></Variable>
                </Assign>
            </Assignment>
            <Call value="A_000036"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_000036"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000054 "></Comment>
            <A_Proc_Call>
                <Name value="FMT001A2"></Name>
                <Expressions>
                    <String value="FMT001A2"></String>
                </Expressions>
                <Lvalues>
                    <Var_Lvalue value="result_code"></Var_Lvalue>
                    <Var_Lvalue value="os"></Var_Lvalue>
                </Lvalues>
            </A_Proc_Call>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r15"></Var_Lvalue>
                    <Variable value="result_code"></Variable>
                </Assign>
            </Assignment>
            <Call value="A_000046"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_000046"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000061 "></Comment>
            <A_Proc_Call>
                <Name value="FMT001A3"></Name>
                <Expressions>
                    <String value="FMT001A3"></String>
                </Expressions>
```

```
                <Lvalues>
                    <Var_Lvalue value="result_code"></Var_Lvalue>
                    <Var_Lvalue value="os"></Var_Lvalue>
                </Lvalues>
            </A_Proc_Call>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r15"></Var_Lvalue>
                    <Variable value="result_code"></Variable>
                </Assign>
            </Assignment>
            <Call value="A_000056"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_000056"></Name>
        <Statements>
            <Comment value="*"></Comment>
            <Comment value="&lt;FermaT&gt; 00000072 WSAVE WSAVE[5]
                WSAVE[6] WSAVE[7] WSAVE[8]"></Comment>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r13"></Var_Lvalue>
                    <Sub_Seg>
                        <Variable value="WSAVE"></Variable>
                        <Number value="5"></Number>
                        <Number value="8"></Number>
                    </Sub_Seg>
                </Assign>
            </Assignment>
            <Call value="A_00005A"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_00005A"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000073 "></Comment>
            <A_Proc_Call>
                <Name value="pop_regs"></Name>
                <Expressions></Expressions>
                <Lvalues>
                    <Var_Lvalue value="r0"></Var_Lvalue>
                    <Var_Lvalue value="r1"></Var_Lvalue>
                    <Var_Lvalue value="r2"></Var_Lvalue>
                    <Var_Lvalue value="r3"></Var_Lvalue>
                    <Var_Lvalue value="r4"></Var_Lvalue>
                    <Var_Lvalue value="r5"></Var_Lvalue>
                    <Var_Lvalue value="r6"></Var_Lvalue>
                    <Var_Lvalue value="r7"></Var_Lvalue>
                    <Var_Lvalue value="r8"></Var_Lvalue>
                    <Var_Lvalue value="r9"></Var_Lvalue>
                    <Var_Lvalue value="r10"></Var_Lvalue>
                    <Var_Lvalue value="r11"></Var_Lvalue>
                    <Var_Lvalue value="r12"></Var_Lvalue>
                    <Var_Lvalue value="r13"></Var_Lvalue>
                    <Var_Lvalue value="r14"></Var_Lvalue>
                    <Var_Lvalue value="reg_stack"></Var_Lvalue>
                </Lvalues>
```

```
            </A_Proc_Call>
            <Call value="A_00005E"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_00005E"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000074 "></Comment>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="r15"></Var_Lvalue>
                    <Number value="0"></Number>
                </Assign>
            </Assignment>
            <Cond>
                <Guarded>
                    <Equal>
                        <Variable value="r15"></Variable>
                        <Number value="0"></Number>
                    </Equal>
                    <Statements>
                        <Assignment>
                            <Assign>
                                <Var_Lvalue value="cc"></Var_Lvalue>
                                <Number value="0"></Number>
                            </Assign>
                        </Assignment>
                    </Statements>
                </Guarded>
                <Guarded>
                    <True></True>
                    <Statements>
                        <Assignment>
                            <Assign>
                                <Var_Lvalue value="cc"></Var_Lvalue>
                                <Number value="2"></Number>
                            </Assign>
                        </Assignment>
                    </Statements>
                </Guarded>
            </Cond>
            <Call value="A_000060"></Call>
        </Statements>
    </Action>
    <Action>
        <Name value="A_000060"></Name>
        <Statements>
            <Comment value="&lt;FermaT&gt; 00000075 "></Comment>
            <Assignment>
                <Assign>
                    <Var_Lvalue value="destination"></Var_Lvalue>
                    <Variable value="r14"></Variable>
                </Assign>
            </Assignment>
            <Call value="dispatch"></Call>
        </Statements>
    </Action>
    <Action>
```

```
                    <Name value="dispatch"></Name>
                    <Statements>
                        <Cond>
                            <Guarded>
                                <Equal>
                                    <Variable value="destination"></Variable>
                                    <Number value="0"></Number>
                                </Equal>
                                <Statements>
                                    <Call value="Z"></Call>
                                </Statements>
                            </Guarded>
                            <Guarded>
                                <True></True>
                                <Statements>
                                    <Comment value="Unknown destination "></Comment>
                                    <Call value="Z"></Call>
                                </Statements>
                            </Guarded>
                        </Cond>
                    </Statements>
                </Action>
            </Actions>
        </A_S>
    </Statements>
</Var>
</Statements>
```

List 9-3. FMT001A0 AST in XML

9.2.3 CML Model Extraction from WSL

Constructing a useful model necessarily involves throwing away some information: in other words, to be useful a model must be inaccurate, or at least idealised, to a certain extent [180]. In Subsection 7.1.2, an extraction algorithm is introduced where human interaction is necessary.

The first step is to collect the data and control information from the source code. List 9-4 and List 9-5 show the data and control information that can be used to calculate the metrics.

```
STARTDCT
  COUNT 11
  MODULE FMT001A0
    FUNCTIONID 5
      STARTSECTION
        SECTIONID 1
        DCDNAME F001A0C1
        STARTDATA
      ELEMENT FMT001A0_CSECT
```

```
        DATAID 1
        TYPE 1
        PARENT 0
        MORE
      ELEMENT R3
        DATAID 2
        TYPE 5
        PARENT 1
        MORE
      ELEMENT R12
        DATAID 3
        TYPE 5
        PARENT 1
        MORE
      ELEMENT R13
        DATAID 4
        TYPE 5
        PARENT 1
        MORE
      ELEMENT R14
        DATAID 5
        TYPE 5
        PARENT 1
        MORE
      ELEMENT R15
        DATAID 6
        TYPE 5
        PARENT 1
        MORE
      ELEMENT WDUB
        DATAID 7
        TYPE 4
        PARENT 1
        MORE
      ELEMENT WSAVE
        DATAID 8
        TYPE 4
        PARENT 1
        MORE
      ELEMENT WCT
        DATAID 9
        TYPE 4
        PARENT 1
        MORE
      ELEMENT WNUM
        DATAID 10
        TYPE 4
        PARENT 1
        MORE
      ELEMENT WTOTAL
        DATAID 11
        TYPE 4
        PARENT 1
    ENDDATA
  ENDSECTION
ENDDCT
```

List 9-4. Data Flow of of Module FMT001A0

```
STARTFCT
  COUNT 65
  MODULE PROJ01
    FUNCTIONID 1
    TYPE 0  EXECLINES 719  COMPLEXITY D
    MCCABE 85  PARENT 0  VERMAJ -1  VERMIN 0
    MORE
    MODULE MODULE
      FUNCTIONID 2
      TYPE 1  EXECLINES 612  COMPLEXITY D
      MCCABE 79  PARENT 1  VERMAJ -1  VERMIN 0
      MORE
    MODULE MACRO
      FUNCTIONID 3
      TYPE 1  EXECLINES 87  COMPLEXITY B
      MCCABE 7  PARENT 1  VERMAJ -1  VERMIN 0
      MORE
    MODULE COPYBOOK
      FUNCTIONID 4
      TYPE 1  EXECLINES 20  COMPLEXITY A
      MCCABE 1  PARENT 1  VERMAJ -1  VERMIN 0
      MORE
      MODULE FMT001A0
        FUNCTIONID 5
        TYPE 3  EXECLINES 25  COMPLEXITY A
        MCCABE 2  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A1
        FUNCTIONID 6
        TYPE 3  EXECLINES 65  COMPLEXITY C
        MCCABE 15  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A2
        FUNCTIONID 7
        TYPE 3  EXECLINES 27  COMPLEXITY A
        MCCABE 2  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A3
        FUNCTIONID 8
        TYPE 3  EXECLINES 68  COMPLEXITY A
        MCCABE 2  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A4
        FUNCTIONID 9
        TYPE 3  EXECLINES 271  COMPLEXITY C
        MCCABE 48  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A5
        FUNCTIONID 10
        TYPE 3  EXECLINES 56  COMPLEXITY A
        MCCABE 5  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A6
        FUNCTIONID 11
        TYPE 3  EXECLINES 55  COMPLEXITY A
        MCCABE 5  PARENT 2  VERMAJ 1  VERMIN 1
        MORE
      MODULE FMT001A7
```

FUNCTIONID 12
TYPE 3 EXECLINES 22 COMPLEXITY A
MCCABE 5 PARENT 2 VERMAJ 1 VERMIN 1
MORE
MODULE FMT200P9
FUNCTIONID 13
TYPE 3 EXECLINES 23 COMPLEXITY A
MCCABE 3 PARENT 2 VERMAJ 1 VERMIN 1
MORE
MODULE DMAC
FUNCTIONID 15
TYPE 4 EXECLINES 11 COMPLEXITY A
MCCABE 1 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE DMAC01
FUNCTIONID 16
TYPE 4 EXECLINES 8 COMPLEXITY A
MCCABE 1 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE DMAC02
FUNCTIONID 17
TYPE 4 EXECLINES 8 COMPLEXITY A
MCCABE 1 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE PRGNTR1
FUNCTIONID 18
TYPE 4 EXECLINES 5 COMPLEXITY A
MCCABE 2 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE PRGNTR2
FUNCTIONID 19
TYPE 4 EXECLINES 5 COMPLEXITY A
MCCABE 2 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE USRMAC1
FUNCTIONID 20
TYPE 4 EXECLINES 4 COMPLEXITY A
MCCABE 1 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE USRMAC2
FUNCTIONID 21
TYPE 4 EXECLINES 16 COMPLEXITY A
MCCABE 3 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE USRMAC3
FUNCTIONID 22
TYPE 4 EXECLINES 12 COMPLEXITY A
MCCABE 2 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE USRMAC4
FUNCTIONID 23
TYPE 4 EXECLINES 4 COMPLEXITY A
MCCABE 1 PARENT 3 VERMAJ 1 VERMIN 1
MORE
MODULE USRMACA
FUNCTIONID 24
TYPE 4 EXECLINES 14 COMPLEXITY A
MCCABE 2 PARENT 3 VERMAJ 1 VERMIN 1

```
    MORE
  MODULE FMT001W1
    FUNCTIONID 14
    TYPE 7  EXECLINES 20  COMPLEXITY A
    MCCABE 1  PARENT 4  VERMAJ 1  VERMIN 1
    MORE
    MODULE GETREC
      FUNCTIONID 32
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 6  VERMAJ -1  VERMIN 0
      MORE
    MODULE FILEOPEN
      FUNCTIONID 53
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 9  VERMAJ -1  VERMIN 0
      MORE
    MODULE INITIAL
      FUNCTIONID 54
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 9  VERMAJ -1  VERMIN 0
      MORE
    MODULE PROC1
      FUNCTIONID 55
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 9  VERMAJ -1  VERMIN 0
      MORE
    MODULE PROC11
      FUNCTIONID 56
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 9  VERMAJ -1  VERMIN 0
      MORE
    MODULE PROC2
      FUNCTIONID 57
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 9  VERMAJ -1  VERMIN 0
      MORE
    MODULE FMTAAA1
      FUNCTIONID 60
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 11  VERMAJ -1  VERMIN 0
      MORE
    MODULE FMTBBB2
      FUNCTIONID 61
      TYPE 8  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 11  VERMAJ -1  VERMIN 0
      MORE
    MODULE FMT501
      FUNCTIONID 64
      TYPE 9  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 12  VERMAJ -1  VERMIN 0
      MORE
    MODULE FMT502
      FUNCTIONID 65
      TYPE 9  EXECLINES  COMPLEXITY
      MCCABE 0  PARENT 12  VERMAJ -1  VERMIN 0
      MORE
  MODULE FMT101
    FUNCTIONID 34
```

```
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT102
  FUNCTIONID 35
  TYPE 12 EXECLINES COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT103
  FUNCTIONID 36
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT104
  FUNCTIONID 37
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT105
  FUNCTIONID 38
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT106
  FUNCTIONID 39
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT107
  FUNCTIONID 40
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT108
  FUNCTIONID 41
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT110
  FUNCTIONID 42
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT111
  FUNCTIONID 43
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT112
  FUNCTIONID 44
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT113
  FUNCTIONID 45
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
```

```
MODULE FMT114
  FUNCTIONID 46
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT115
  FUNCTIONID 47
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT300
  FUNCTIONID 48
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT301
  FUNCTIONID 49
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT302
  FUNCTIONID 50
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT303
  FUNCTIONID 51
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE UNKNOWN
  FUNCTIONID 52
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT401
  FUNCTIONID 58
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT402
  FUNCTIONID 59
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT601
  FUNCTIONID 62
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE FMT602
  FUNCTIONID 63
  TYPE 12  EXECLINES  COMPLEXITY
  MCCABE 0  PARENT 2  VERMAJ -1  VERMIN 0
  MORE
MODULE REGEQU
  FUNCTIONID 26
  TYPE 13  EXECLINES  COMPLEXITY
```

```
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE DXPUT
        FUNCTIONID 33
        TYPE 13   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE CALL
        FUNCTIONID 25
        TYPE 14   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE CLOSE
        FUNCTIONID 27
        TYPE 14   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE DCB
        FUNCTIONID 28
        TYPE 14   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE GET
        FUNCTIONID 29
        TYPE 14   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE OPEN
        FUNCTIONID 30
        TYPE 14   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
        MORE
      MODULE PUT
        FUNCTIONID 31
        TYPE 14   EXECLINES   COMPLEXITY
        MCCABE 0   PARENT 3   VERMAJ -1   VERMIN 0
ENDFCT
```

List 9-5. Control Flow of Package

9.2.4 Mapping to UML

In Subsection 7.4.1, mapping from CML to UML is introduced. List 9-6 shows the result of generated UML in XMI format.

```
<?xml version="1.0" standalone ="yes"?>
<XMI xmi.version="1.1" xmlns:UML="omg.org/UML/1.4">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>ru.novosoft .uml.impl .UMLRepositoryImplXMIWriter
      </XMI.exporter>
```

```
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <UML:CallState xmi.id="a0" name="FMT001A0 (Cont.)"
      isSpecification =" false " isDynamic="false" outgoing="a11"
      incoming="a12"></UML:CallState>
    <UML:CallState xmi.id="a1" name="FMT001A0 (Cont.)"
      isSpecification =" false " isDynamic="false" outgoing="a13"
      incoming="a14"></UML:CallState>
    <UML:CallState xmi.id="a10" name="FMT001A0 (entry)"
      isSpecification =" false " isDynamic="false" outgoing="a15"
      incoming="a16"></UML:CallState>
    <UML:DataType xmi.id="a2" isRoot="false" isLeaf=" false "
      isAbstract =" false " name="byte" isSpecification =" false ">
    </UML:DataType>
    <UML:CallState xmi.id="a3" name="FILEOPEN (Entry)"
      isSpecification =" false " isDynamic="false" outgoing="a17"
      incoming="a13"></UML:CallState>
    <UML:CallState xmi.id="a4" name="FMT001A2 (Entry)"
      isSpecification =" false " isDynamic="false" outgoing="a18"
      incoming="a11"></UML:CallState>
    <UML:CallState xmi.id="a5" name="FMT001A1 (Entry)"
      isSpecification =" false " isDynamic="false" outgoing="a19"
      incoming="a15"></UML:CallState>
    <UML:CallState xmi.id="a6" name="FMT001A1 (Return)"
      isSpecification =" false " isDynamic="false" outgoing="a12"
      incoming="a19"></UML:CallState>
    <UML:Package xmi.id="a7" isRoot="true" isLeaf=" false "
      isAbstract =" false " name="Test" isSpecification =" false ">
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id="a20" isRoot="false" isLeaf=" false "
          isAbstract =" false " name="FMT001A0"
          isSpecification =" false " isActive =" false ">
          <UML:Namespace.ownedElement>
            <UML:Attribute xmi.id="a21" name="WDS1"
              isSpecification =" false " type="a2"></UML:Attribute>
            <UML:Attribute xmi.id="a22" name="WDS1.WDS1F1"
              isSpecification =" false " type="a2"></UML:Attribute>
```

```
        <UML:Attribute xmi.id="a23" name="WDS1.WDS1F2"
          isSpecification =" false " type="a2"></UML:Attribute>
        <UML:Attribute xmi.id="a24" name="WDS1.WDS1F3"
          isSpecification =" false " type="a2"></UML:Attribute>
        <UML:Attribute xmi.id="a25" name="WDS1.WDS1F4"
          isSpecification =" false " type="a2"></UML:Attribute>
      </UML:Namespace.ownedElement>
      <UML:Classifier. feature >
        <UML:Operation xmi.id="a26" isRoot="false" isLeaf=" false "
          isAbstract =" false " isQuery=" false " name="FILEOPEN"
          visibility ="public" isSpecification =" false ">
        </UML:Operation>
      </UML:Classifier . feature >
    </UML:Class>
    <UML:Class xmi.id="a27" isRoot="false" isLeaf=" false "
      isAbstract =" false " name="FMT001A1" isSpecification="false"
      isActive =" false "></UML:Class>
    <UML:Class xmi.id="a28" isRoot="false" isLeaf=" false "
      isAbstract =" false " name="FMT001A2" isSpecification="false"
      isActive =" false "></UML:Class>
    <UML:Class xmi.id="a29" isRoot="false" isLeaf=" false "
      isAbstract =" false " name="FMT001A3" isSpecification="false"
      isActive =" false "></UML:Class>
    <UML:Association xmi.id="a30" isRoot="false " isLeaf=" false "
      isAbstract =" false " name="" isSpecification =" false ">
      <UML:Association.connection>
        <UML:AssociationEnd xmi.id="a31" isSpecification =" false "
          isNavigable=" false " participant ="a20">
        </UML:AssociationEnd>
        <UML:AssociationEnd xmi.id="a32"
          isSpecification =" false " isNavigable="true "
          participant ="a27">
        </UML:AssociationEnd>
      </UML:Association.connection>
    </UML:Association>
    <UML:Association xmi.id="a33" isRoot="false " isLeaf=" false "
      isAbstract =" false " name="" isSpecification =" false ">
      <UML:Association.connection>
```

```
        <UML:AssociationEnd xmi.id="a34" isSpecification =" false "
          isNavigable=" false " participant ="a20">
        </UML:AssociationEnd>
        <UML:AssociationEnd xmi.id="a35" isSpecification =" false "
          isNavigable="true " participant ="a28">
        </UML:AssociationEnd>
      </UML:Association.connection>
    </UML:Association>
    <UML:Association xmi.id="a36" isRoot="false " isLeaf=" false "
      isAbstract =" false " name="" isSpecification =" false ">
      <UML:Association.connection>
        <UML:AssociationEnd xmi.id="a37" isSpecification =" false "
          isNavigable=" false " participant ="a20">
        </UML:AssociationEnd>
        <UML:AssociationEnd xmi.id="a38" isSpecification =" false "
          isNavigable="true " participant ="a29">
        </UML:AssociationEnd>
      </UML:Association.connection>
    </UML:Association>
    <UML:ActivityGraph xmi.id="a39" isSpecification =" false ">
      <UML:ActivityGraph.partition>
        <UML:Partition xmi.id="a40" contents ="a9 a10 a0 a1 a3 a41"
          name="FMT001A0" isSpecification="false">
        </UML:Partition>
        <UML:Partition xmi.id="a42" contents ="a5 a6"
          name="FMT001A1" isSpecification="false">
        </UML:Partition>
        <UML:Partition xmi.id="a43" contents ="a4 a8"
          name="FMT001A2" isSpecification="false">
        </UML:Partition>
      </UML:ActivityGraph.partition >
      <UML:StateMachine.top>
        <UML:CallState xmi.id="a41" name="FILEOPEN (Return)"
          isSpecification =" false " isDynamic="false"
          incoming="a17">
        </UML:CallState>
      </UML:StateMachine.top>
      <UML:StateMachine.transitions>
```

```
                <UML:Transition xmi.id="a16" source="a9" target ="a10"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a15" source="a10" target ="a5"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a19" source="a5" target ="a6"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a12" source="a6" target ="a0"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a11" source="a0" target ="a4"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a18" source="a4" target ="a8"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a14" source="a8" target ="a1"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a13" source="a1" target ="a3"
                  name="" isSpecification =" false "></UML:Transition>
                <UML:Transition xmi.id="a17" source="a3" target ="a41"
                  name="" isSpecification =" false "></UML:Transition>
              </UML:StateMachine. transitions>
            </UML:ActivityGraph>
          </UML:Namespace.ownedElement>
        </UML:Package>
        <UML:CallState xmi.id="a8" name="FMT001A2 (Return)"
          isSpecification =" false " isDynamic="false" outgoing="a14"
          incoming="a18"></UML:CallState>
        <UML:CallState xmi.id="a9" name="Start" isSpecification =" false "
          isDynamic="false" outgoing="a16"></UML:CallState>
      </XMI.content>
    </XMI>
```

List 9-6. Generated UML in XMI File

9.2.5 Visualisation and Redocumentation

Currently, two diagrams are extracted, UML Class Diagram and UML Activity Diagram. The static view of the class diagram represents a mixture of call graphs, data catalogues and function catalogues. The dynamic view of the activity diagram on the other hand shows the interaction among the legacy system components/modules. Figure

9-2 shows a result generated by F-UML, where some classes are grouped into a package. The final generated documentation as shown in Figure 9-3 is browser based and easy navigation. Since all the information in repository is XML based, only representation style templates are required for HTML generation.

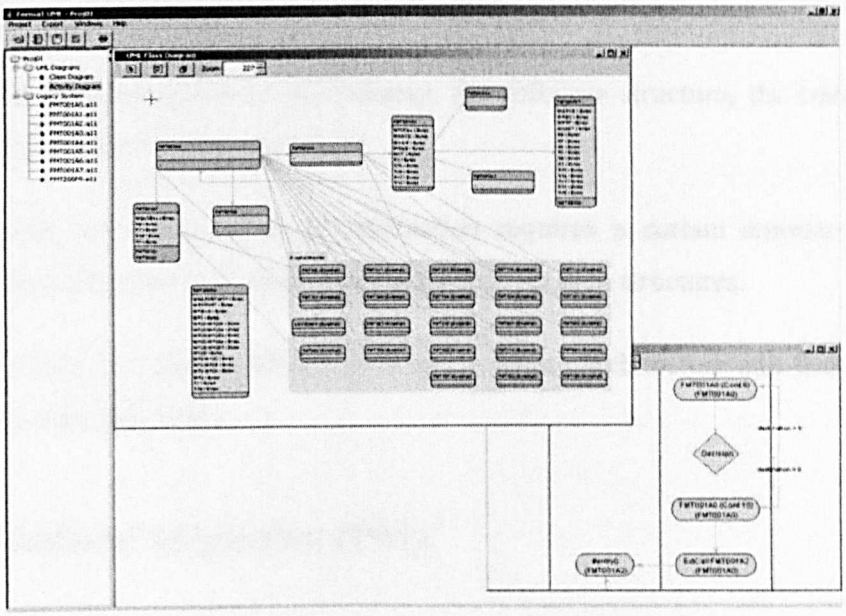


Figure 9-2. F-UML Presentation Tool

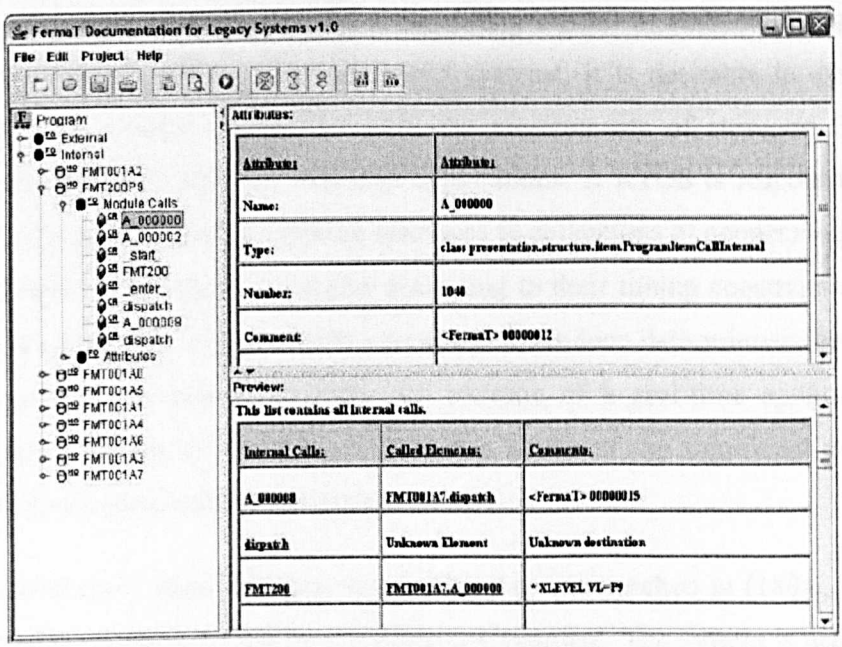


Figure 9-3. GUI for Presentation Layer

9.2.6 Discussion

This case study is a particularly challenging reverse engineering task: to extract high-level models from IBM 370 assembler programs. The obtained results reveal that:

(1) The abstracted entities are hard to be named.

(2) Since the transformation will change the software structure, the traceability of transformation is broken in some ways.

(3) Moving to higher levels of abstraction requires a certain amount of human intervention, particularly to select appropriate abstract data structures.

(4) Since this is a small software package, software architecture and feature are not discussed in this case study.

9.3 Platform Migration (PM)

9.3.1 Background

Real-Time Operating System (RTOS) is becoming crucial in embedded system. With the ever increasing complexity of embedded systems, it is desirable to employ Real Time Operating System (RTOS) to fulfil the requirements of stringent timing and resource constraints of different real-time applications. A RTOS is responsible for the allocation of processors and computing resources to collections of cooperating tasks in a way which will enable them to execute according to their timing constraints. A RTOS provides the developer with the tools necessary to produce deterministic behaviour in the final system and hence facilitates the creation of a real-time application [99]. Software migration on different target platforms is one of the significant problems in RTOS-based software evolution domain.

The selected case, mine drainage system, has been researched in [180] , which is a simplified pump control system for a mining environment. The system is used to pump mine water, which collects in a sump at the bottom of the shaft, to the surface. The main

safety requirement is that the pump should not be operated when the level of methane gas in the mine reaches a high value due to the risk of explosion. Such system was first implemented in RTLinux environment previously, which needs to be run on ThreadX platform now, namely, the software migration from RTLinux to ThreadX. It is a good demonstration of the RTOS specific software migration via the proposed approach.

9.3.1.1 Overview of RTOS-Specific Software Migration

Figure 9-4 demonstrates two often-happened situations in RTOS specific software migration, the software migration between two different platforms, e.g., from RTLinux to ThreadX.

- In the first situation (Figure 9-4A), when application software is migrated from one RTOS platform to another one, system APIs will be the crucial part of this migration process. Different RTOS platform provides different APIs. Developers can define program transformation rules and transform application software based on their knowledge of different platform APIs. Since the POSIX standard contains most of the standard UNIX compatible system call interface, many RTOS platforms support subset of POSIX standard.

- In the second situation (Figure 9-4B), Virtual Operating System (VOS) plays a role as a middleware which is running on Windows platform but supporting different RTOSs' programming so that application software developed on VOS can be ported to target RTOS platform directly without any change.

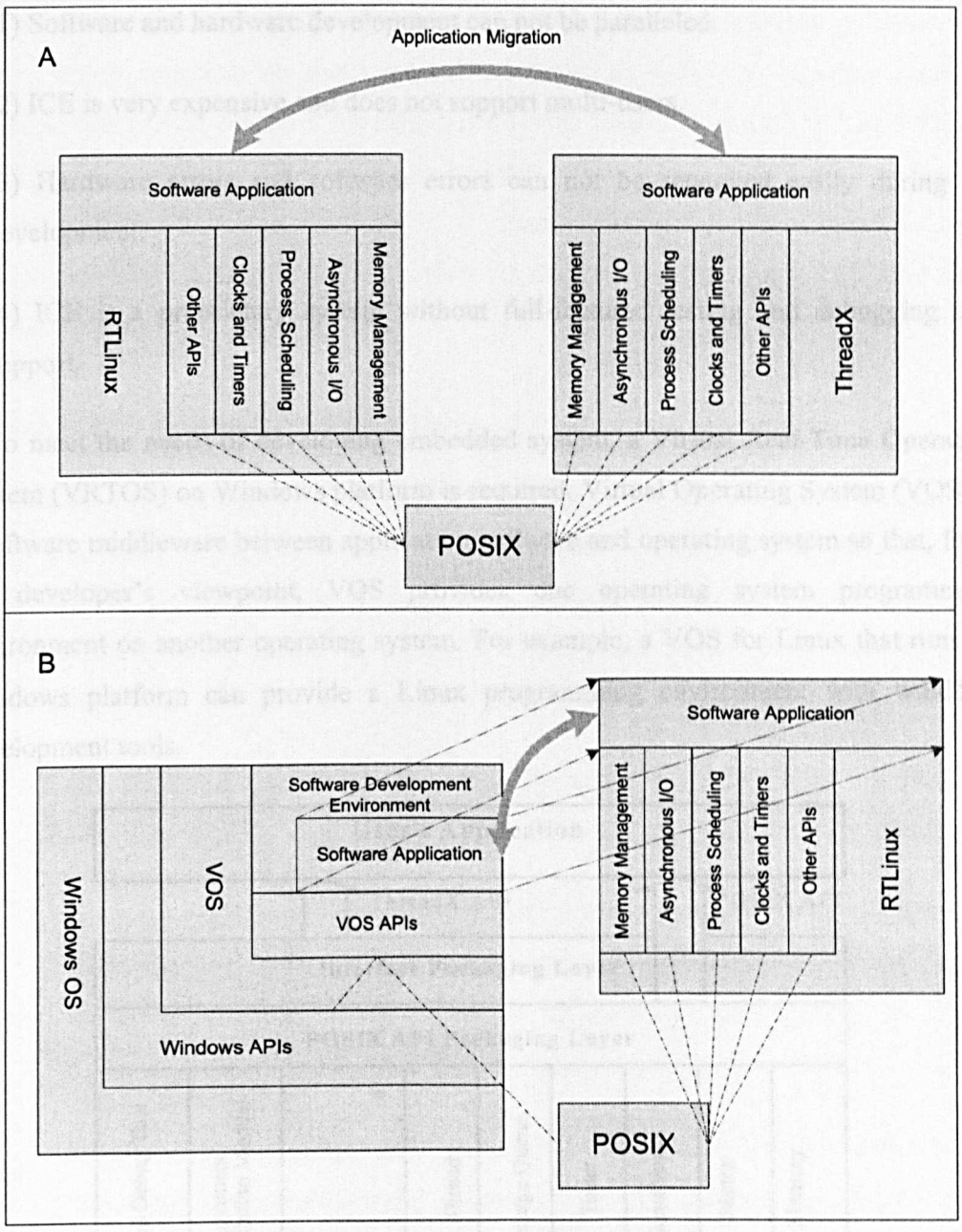


Figure 9-4. RTOS Specific Software Migration

9.3.1.2 VOS and VRTOS

Even though the embedded system development is supported by RTOS, the general development environment is In-Circuit Emulator (ICE) that has some disadvantages for software development:

(1) Software and hardware development can not be paralleled.

(2) ICE is very expensive and does not support multi-users.

(3) Hardware errors and software errors can not be separated easily during the development.

(4) ICE is a proprietary system without full-featured testing and debugging tool support.

To meet the needs of developing embedded system, a Virtual Real-Time Operating System (VRTOS) on Windows platform is required. Virtual Operating System (VOS) is a software middleware between application software and operating system so that, from the developer's viewpoint, VOS provides one operating system programming environment on another operating system. For example, a VOS for Linux that runs on Windows platform can provide a Linux programming environment with windows development tools.

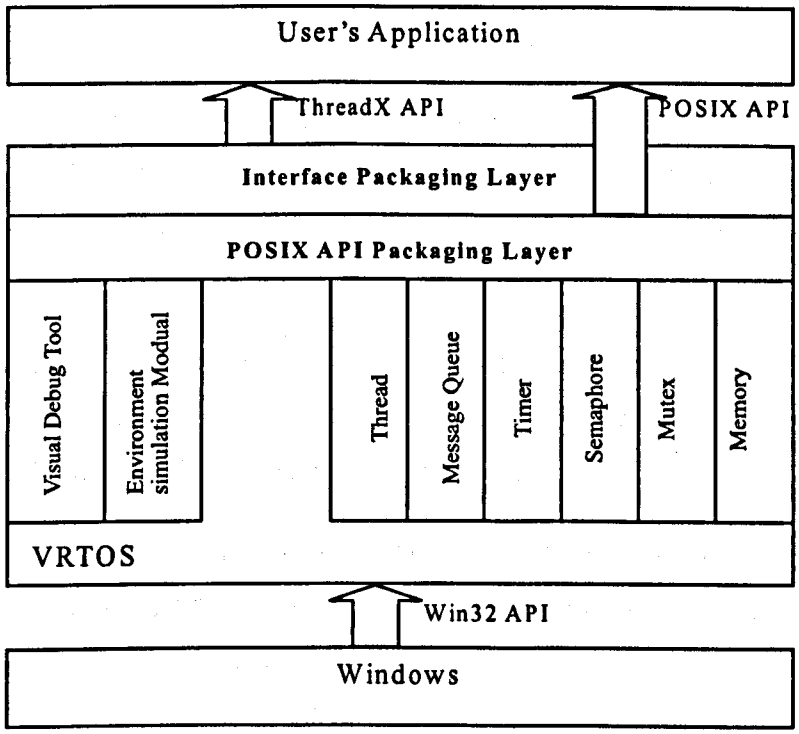


Figure 9-5. Architecture of VRTOS

The architecture of VRTOS is shown in Figure 9-5. VRTOS provides a *Kernel API Layer*, which supports the real-time POSIX Standard [66]. An *Interface Layer* is designed that can be extended for different RTOSs, such as ThreadX [37] or RTLinux [138]. VRTOS supports the pre-emptive schedule policy of the First-Come-First-Served (FCFS) style and simulates many kinds of system resources. A visual debug tool that enables external environment simulation facilitates the debugging of embedded software greatly [162].

9.3.1.3 Model Driven Migration Issues

To meet the requirement of RTOS specific software migration, a platform independent model of RTOS is defined using the MDA approach, which RTOS functions and properties can be specified with a platform independent model. The methodology adopts interface specifications as the main specification technique for the following three reasons [61]:

- Firstly, interfaces can be relatively cheaply derived from both system models and legacy systems.
- Secondly, interfaces can be used to isomorphically represent both system models (PIMs), that are architected during the forward engineering phase, and legacy wrappers are acquired during the reverse engineering phase (PSMs), specifying the abstract services implemented by the legacy systems.
- Thirdly, once mappings are established, they can straightforwardly be transformed into parameterised components (using standard MDA mappings) that rely for part of their execution on legacy wrappers.

The migration task between two RTOSs can be performed between RTOS and VRTOS firstly since VRTOS can provide a programming environment on Windows platform that the target code can be debugged on VRTOS. After debugging, the target code can be ported to physical RTOS without any change. The following Sections show the main processes of REMOST approach that is effective and time-saving.

9.3.2 RTOS Feature Model

In this case study, an initial set of RTOS feature models are built first, which are extracted based on software developer's knowledge about RTOS modelling domain. The feature model will be conducted through reading related documentation. In this case study, source RTOS is RTLinux and target RTOS is ThreadX. Feature model of RTLinux and ThreadX needs to be created.

During the software migration from one RTOS platform to another one, system APIs will be the crucial part of this migration process. POSIX standard is selected as starting point for RTOS interface specification. List 9-7 shows part of a feature model in FDL based on POSIX standard.

```
Posix: All (
   PosixAPI,
   PosixService,
   ...
   )
   ...
PosixAPI: All (
   Thread,
   Timer,
   Message Queue
   Semaphore,
   Memory,
   )
   ...
Thread: All (
   Mutex,
   Scheduling,
   ...
   )
   ...
Mutex All (
   pthread_mutex_init()
   pthread_mutex_destroy()
   pthread_mutex_getprioceiling()
   pthread_mutex_lock()
   pthread_mutex_setprioceiling()
   pthread_mutex_timedlock()
   pthread_mutex_trylock()
   pthread_mutex_unlock()
   )
   ...
PosixService All (
   Scheduling_policies,
```

```
    ...
    )
    ...
  Scheduling_policies Oneof (
    SCHED_FIFO,
    SCHED_OTHER,
    SCHED_RR,
    SCHED_SPORADIC,
    )
    ...
```

List 9-7. Part of POSIX Standard Feature Model

From the feature model, developers can get the required information quickly and precisely. In this case study, if developers are willing to get the information about existing POSIX APIs that are defined in both systems providing thread creating service, they can query knowledge base by retrieving the instance of APIs which are defined in RTLinux feature model and ThreadX feature model. If the query result suggests that both systems have POSIX API implemented to create thread, then that part of application can be migrated by changing API's names directly. If the query result shows that not both systems have such POSIX API, then the API will need to be re-implemented during the migration. Likewise, if one platform features are not supported by other platform, corresponding development is required.

From the RTLinux application source code, particular system APIs are extracted and matched with ThreadX feature model. A series of transformation rules are defined in List 9-8.

```
Rule Set 1: ANSI_C2ANSI_C API Transformation
  Preconditions:
    1. assert(SourceAPI.isInstance(true)), and
    2. assert(SourceAPI.hasAPIStandard.equals(POSIX)), and
    3. assert(SourceAPI.getRDFType().equals (ANSI_C_api));
  Transformation rules:
    TargetAPI.all()=SourceAPI.all();
Rule Set 2: POSIX2POSIX API Transformation
  Preconditions:
    1. assert(SourceAPI.isInstance(true)), and
    2. assert(TargetAPI.isInstance(true)), and
    3. assert(SourceAPI.hasAPIStandard.equals(POSIX)), and
    4. assert(TargetAPI.hasAPIStandard.equals(POSIX)), and
    5. assert(!SourceAPI.getRDFType().equals(ANSI_C_api));
  Transformation rules:
    TargetAPI.all()=SourceAPI.all();
```

Rule Set 3, 4 and 5:
POSIX2NONPOSIX, NONPOSIX2POSIX,NONPOSIX2NONPOSIX
API Transformation
Preconditions:
1. assert(SourceAPI.isInstance(true)), and
2. assert(TargetAPI.isInstance(true)), and
3. (assert(SourceAPI.hasAPIStandard.equals(POSIX)), and
assert(TargetAPI.hasAPIStandard.equals (NONPOSIX))),
Or (assert(SourceAPI.hasAPIStandard.equals (NONOSIX)), and
assert(TargetAPI.hasAPIStandard.equals(POSIX))),
Or (assert(SourceAPI.hasAPIStandard.equals (NONPOSIX)), and
assert(TargetAPI.hasAPIStandard.equals (NONPOSIX))), and
4. assert(!SourceAPI.getRDFType().equals (ANSI_C_api));

Transformation rules:
Replace target API with source API.

List 9-8. Feature Based Transformation Rules

Based on above transformation rules, RTLinux POSIX/ANSI C API can be transformed into ThreadX POSIX/ANSI C API directly, such as printf() and srand(). RTLinux POSIX API can be transformed into ThreadX NONPOSIX API with the instructions of specific transformation rules, e.g., from pthread_create() to tx_thread_create(), as well as from malloc() to tx_byte_pool_create().

9.3.3 RTOS Specific Program Transformation

The source code needs translated into WSL and processed as before, which will not be discussed in this case study. Here only system APIs will be translated with the help of feature model. List 9-9 shows a slice of program source code developed on RTLinux platform. List 9-10 shows the transformed target code for ThreadX platform.

```
void* ThreadProc(void* para)
{
#define MAX_BLOCK    6000
void * *pMem[MAX_BLOCK];
unsigned long ulTotalSize = 0;
int memsize, i ;

printf ("Thread %u run:\n", (ULONG) para);
    printf ("parameter is %u\n",para);
    srand ( (unsigned) time ( NULL ) );
    for ( i = 0 ; i < MAX_BLOCK; i ++ ) {
        memsize = rand () % (1024*1024*4);
        pMem[i] = malloc (memsize);
```

```
            printf ("mem[%d] = 0x%-8X size = %d\n", i, pMem[i], memsize);
            if ( pMem[i] == NULL ) {
                printf ("alloc count:%d   need size = %d   left size = %d\n",
                        i, memsize, TOTAL_MEM - ulTotalSize);
                getch ();
                break;
            }
            else {
                ulTotalSize += memsize;
            }
        }
        if ( i > 0 && pMem[i] == NULL ) i --;
        while ( i >= 0 && pMem[i] != NULL ) {
            free (pMem[i]);
            printf ("free mem[%d] = 0x%-8X \n", i, pMem[i]);
            i --;
        }
        printf ("Thread %u finish.\n", (ULONG) para);
        pthread_cancel (pthread_self ());
        return 0;
}

void main ()
{
        pthread_attr_t   attr;
        pthread_t   ThreadId1;
        MsgQueID = mq_open ("Thread1MessageQueue",
                                O_CREAT | O_APPEND, 0, NULL);
        pthread_attr_init (&attr);
        pthread_attr_setschedpolicy (&attr, SCHED_FIFO);
        pthread_create (&ThreadId1, &attr, ThreadProc, (void *) 1);
        pthread_setschedprio (ThreadId1, 5);
        pthread_attr_destroy (&attr);
}
```

List 9-9. Source Code on RTLinux

```
void* ThreadProc (void* para)
{
#define MAX_BLOCK     6000
        void * *pMem[MAX_BLOCK];
        unsigned long ulTotalSize = 0;
        int memsize, i ;
        TX_BYTE_POOL byte_pool_0;

        printf ("Thread %u run:\n", (ULONG) para);
        printf ("parameter is %u\n", para);
        srand ( (unsigned) time ( NULL ) );
        for ( i = 0 ; i < MAX_BLOCK; i ++ ) {
            memsize = rand () % (1024*1024*4);
            tx_byte_pool_create (&byte_pool_0, "byte pool 0",
```

```
                                first_unused_memory, DEMO_BYTE_POOL_SIZE);
        pMem[i] = tx_byte_allocate (&byte_pool_0, &pointer,
                                    DEMO_STACK_SIZE, TX_NO_WAIT);
        printf ("mem[%d] = 0x%-8X size = %d\n", i, pMem[i], memsize);
        if ( pMem[i] == NULL ) {
            printf ("alloc count:%d   need size = %d   left size = %d\n",
                    i, memsize, TOTAL_MEM - ulTotalSize);
            getch ();
            break;
        }
        else {
            ulTotalSize += memsize;
        }
    }
    if ( i > 0 && pMem[i] == NULL ) i --;
    while ( i >= 0 && pMem[i] != NULL ) {
        tx_byte_realease(pMem[i]);
        printf ("free mem[%d] = 0x%-8X \n", i, pMem[i]);
        i --;
    }
    printf ("Thread %u finish.\n", (ULONG) para);
    pthread_cancel (pthread_self ());
    return 0;
}

void main()
{
    TX_THREAD   ThreadId1;
    tx_queue_create (&MsgQueID, "Thread1MessageQueue", 0, NULL);
    tx_thread_create (&ThreadId1, "thread 1", ThreadProc, 0, pointer,
                      DEMO_STACK_SIZE, 1, 1,
                      TX_NO_TIME_SLICE, TX_AUTO_START);
    tx_thread_priority_change (&ThreadId1, 5, NULL);
}
```

List 9-10. Target Code on ThreadX

The case, mine drainage system, contains 30 ".c" and ".h" files. Table 9-3 shows the result of the experiment, 4 RTLinux POSIX APIs that appear 48 times in the application are successfully transformed based on transformation rules, as well as 4 RTLinux NONPOSIX APIs which appear 22 times in the application. 29 POSIX/ANSI C APIs can also be transformed under the rules, with total appearance of 108 times. 1 RTLinux NONPOSIX API with 8 appearances cannot be transformed, since there is no proper transformation rule.

API Transformation Statistics

Result	rule-based		manual	
Standard	API	Appearance	API	Appearance
POSIX	4	48	0	0
NONPOSIX	4	22	1	8
ANSI C	29	108	0	0
Total	37	178	1	8

Table 9-3. Migration Statistic

9.3.4 Discussion

Software migration is inherently knowledge intensive, which requires a large number of domain knowledge including system knowledge as well as expertise and experience from specialists. Adding knowledge dimension to software migration approach will be a feasible way to facilitate software migration process by making it more efficiently and accurately. Feature model is proposed to provide understandability, specification, reusability, knowledge acquisition and reliability for software migration.

The experiment results indicate that usage of feature model in software migration is a practically significant. Although twenty percent of APIs still need be transformed manually, the result of this experiment shows that the proposed approach can facilitate software migration greatly. Meanwhile, the experiment result shows that the transformed target code can be running correctly on ThreadX platform. However, it is believed that the proposed approach had additional successes as well as some challenges: Although REMOST approach is based on MDA concepts, the integration of MDA and feature model is still in the preliminary stage; feature model can only be used in special domain, more general approach needs to be proposed to widen its range for utilising; the transformation rules are still very weak.

9.4 Software Modernisation for Agent-based Web Services

9.4.1 Background

Today, Web services are emerging as the new "standard" architectural style. This new architectural style and the software lifecycle it implies are extremely attractive because they can effectively address the demands for short development cycles, distributed development and global user base, at the same time. The new applications developed in this style effectively reuse existing software assets to provide new complex value-added services, fast [151]. The growing demands of Business-to-Business collaborations, for dynamically integrating heterogeneous systems with minimum cost, have fuelled this challenge. In order for this vision of effective reuse-based development to become a reality, a wide base of available Web services is required.

Most projects use the wrapper technical to encapsulate the legacy system into Web services. Wrappers are mechanisms for introducing new behaviour to be executed before, after, in, and/or around an existing method or component [181]. In Chapter 8, WSW tool has been introduced for this purpose. But even for wrapping a session-oriented object as a Web service. It fails to address several significant issues, including those involving object life cycle, object references, and fault handling [158]. To address this problem, AgenEvo project was proposed with an agent-based Web services architecture.

Through agentifying the legacy system, a migration path to allow smooth evolution of agent-based Web services with interaction abilities is provided. This approach is modestly invasive but offers a high return on investment for legacy assets. The architecture of AgenEvo project is illustrated in Figure 9-6, which consists of an Administrator, Accompanying Agents, Mapping Repository and Web services interfaces.

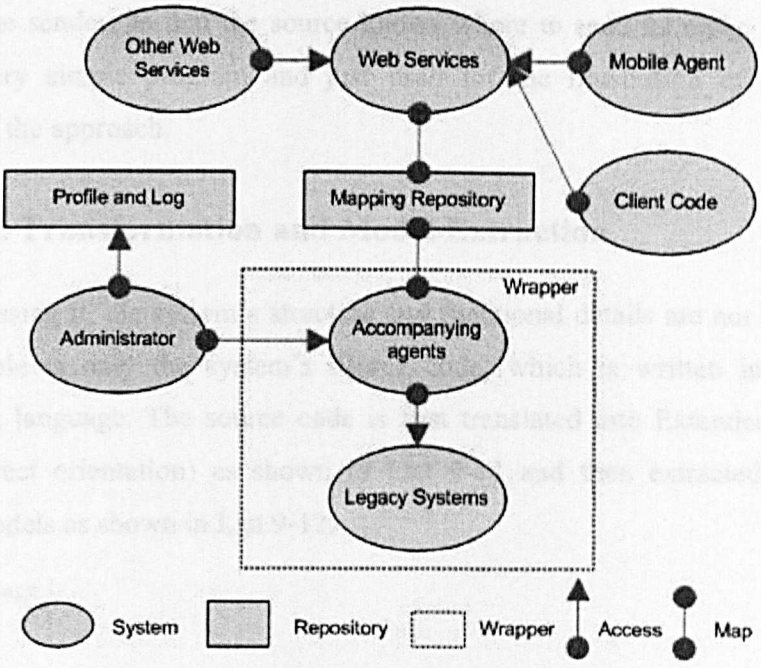


Figure 9-6. Agent-based Web Services Architecture

The Administrator works as an agent server and gets consumers' Web Services invocations. It looks up the Mapping Repository to get the corresponding accompanying agents to create the corresponding Web Services. Furthermore, it is essential for Administrator to know the Web Services execution results and log the invoking history. The accompanying agent is responsible for mapping between the various actors within the model and functions independently of the system. The agent instance executes the appropriate plan or task according to Web Service invoking parameters and depending on the service provider's circumstances. The agent would interact with provided Web Services autonomously. The improper Web service invocation could result in that nothing can be invocated by Administrator.

An example program is used to illustrate AgenEvo project, which is a small size FORTRAN-KCSP program named task farm that has been researched in [97]. In this task farm, every worker communicates directly with the source in order to get jobs and forward results. All workers run their dispatched tasks in parallel. Since the source has no way of telling when a worker has finished its job, and needs another, workers must send requests for more work to the source. These requests must be tagged in some way

to identify the sender, so that the source knows where to send its reply. The example here is a very simple program and just used for the illustration of the technical possibility of the approach.

9.4.2 WSL Transformation and Model Extraction

Before processing it, the system's structure and functional details are not known at all. What available is only the system's source code, which is written in old fashion programming language. The source code is first translated into Extended WSL (with type and object orientation) as shown in List 9-11 and then extracted to the final high-level models as shown in List 9-12.

```
struct message {
   id;
   mbody;
   sender;
   receiver
   };

proc root()
{
   comment:"setup";
   id:=!xf getprocessid();
   !xp mpsynch();
   numw:=!xf mp-grp-size(workerGrp);
   for i:=1 to 100 step 1 do
      buffer[i]:=-1
   od;

   comment:"assign tasks";
   for i:=1 to numw step 1 do
      read t,msg1 from connect[i];
      if (msg1.id <> buffer[i]) and (msg1.receiver=id)
      then buffer[i]:= msg1.id;
         if msg1.mbody="finished"
         then !xp gettask(taskid);
            if tasked < 0
            then msg2.mbody:="idle"
            else msg2.mbody:=!xf strcat("do",taskid)
            fi
         fi;
         if msg1.mbody="faulty"
         then msg2.mbody:="terminate"
         fi;
         msg2.id:=!xf gen-msg-id();
         msg2.sender:=id;
```

```
            msg2.receiver:=i;
            write msg2 to connect[i]
        fi
    od;
    comment:"tide up";
    msg1.id:=0; msg1.mbody:="
};

proc worker()
{
    comment:"setup";
    id:=!xf getprocessid();
    !xp mpsynch();
    buffer:=-1;
    comment:"deal task";
    read t,msg1 from connect[id];
    if (msg1.id <> buffer) and (msg1.receiver=id)
    then buffer:=msg1.id;
        if msg1.mbody="terminate"
        then !xp terminate(taskid)
        fi;
        if msg1.mbody="do xxxxx"
        then taskid:="xxxxx";
            !xp execute(taskid)
        fi;
        if msg1.mbody="idle"
        then skip fi;
    fi;

    comment:"sending message to root";
    taskstate:=!xf gettaskstate(taskid);
    if taskstate="finished"
    then msg2.mbody:="finished" fi;
    if taskstate="faulty"
    then msg2.mbody:="faulty" fi;
    if taskstate="running"
    then skip fi;
    msg2.id:=!xf gen-msg-id();
    msg2.sender:=id;
    msg2.receiver:=0;
    write msg2 to connect[id]
};

proc main()
{
    !xp createprocess(root());
    for i:=1 to W step 1 do
        !xp createprocess(worker())
    od;
};
```

List 9-11. Task Farm in Extended WSL

```
model TASK_FARM
  package aPackage is
    activity Example
      composite state stateMain
        composite state worker
          partitioned in worker
          state RequestService end
          state Finish end
          state Process end
          composite state Root
            partitioned in Root
              state Rtsetup end
              state AssignTasks end
              state tideUp end
              end
          transition Initial
            from worker::RequestService to Root::AssignTasks
          transition
            from Root::AssignTasks to worker::Process
          transition
            from worker::Process to worker::Finish
          transition End
            from worker::Finish to worker::RequestService
    end -- activity Example
  end – package aPackage
end – model TASK_FARM
```

List 9-12. Task Farm Extracted Model in CML

The extracted models gives software engineers a clear impression of the task farm system. It describes the system in a hierarchical order, and only keeps the basic descriptions. From main, it is easy to know that the system consists of one root process and M worker processes. The root process first sets up itself, then assigns tasks to every worker who sends a 'finishes' signal to the root. After the assignment, root resets the message body. For each worker process, it first sets up itself, then processes the assigned task and sends corresponding messages to the root. The root process and all worker process run in parallel.

9.4.3 Architecture Recovery with AADL

Suppose the user want to develop a new agent based Web services system, where legacy system will be rewritten and "AssignTasks()" function as a Web services will be published.

```
Architecture task farm is
{
  AgentTypeList
    root, worker
  AgentType root is
    State
      Numw,connect,id,buffer
    Port
      msg1,msg2
    Services
      requires;
      provides AssignTasks;
      Tasks
        Rtsetup();
        AssignTasks();
        tideUp();
      ...
  Agent_instances are
    root1 instantiates root;
    worker1 instantiates worker;
    ...;
    workerm instantiates worker;
    ...
}
```

List 9-13. Task Farm Software Architecture in ADL

According to the proposed approach, the agent based architecture will be expressed in ADL as illustrated in List 9-13. The Root has one agent instance, which can provide the AssginTasks() services. AssignTasks(numw, msg1, msg2, connect, id, buffer) shows the interface of the service and the detail description of AssginTasks() can be find in the specification. Reverse engineering a program into ADL models is still quite immature. To date, this research does not intend to present a full fledged solution, but to establish a reference model consisting of description languages and a reengineering process. The above process is done manually.

9.4.4 Discussion

Web Services and agent techniques can be brought together in re-engineering the legacy system by providing migration paths to allow smooth evolution. AgenEvo opens up new areas of research. The research in this direction is quite recent and far from resulting into a completely automatic transformation process.

9.5 Summary

Three selected case studies were used to demonstrate that the REMOST approach and FIP toolset can help modernisers perform software modernisation tasks on different sorts of legacy systems more systematically and cost-effectively. Each case study is designed to investigate specific research questions, focusing on different claims.

- The Assembler Migration (AM) case study helps to illustrate detailed steps for using REMOST method.
- The Platform Migration (PM) case study focuses on validating the low cost claim that RTOS specific software could be migrated on a new platform effectively.
- The AgenEvo case study illustrates how to modernise legacy system into state-of-the-art technology, e.g., Agent and Web services.

Chapter 10
Conclusion

In this chapter, the strengths and significance of major results are summarised, an evaluation is provided by revisiting the questions, hypothesis, contributions and success criteria put forward in Chapter 1, the weaknesses and limitations of the work are discussed, and finally, areas for future work are identified.

10.1 Summary of Thesis

This thesis is an attempt to reduce some of the problems inherent with legacy system modernisation through model construction and transformation. The proposed model driven re-engineering approach, REMOST, is a large and difficult effort, which involves many issues related to program and model transformation. Although program transformation and model transformation for software re-engineering has been invested by researchers for some years, dedicated approaches combining program and model transformation have been non-existing.

In this thesis, models are constructed by static analysis of legacy code, which is the only possible system artifact. Once a series of models have been generated, these models can serve as a guide in the redevelopment of this system to a new platform. Code generation with MDA forward engineering tools could automate this task and therefore provide round-trip re-engineering for a legacy system. The supporting tools based on the approach are developed to speed and to scale up practical re-engineering. The case studies confirm that the proposed approach is usefulness, effectiveness and completeness.

10.2 Significance of Contributions and Evaluation

This thesis proposed solutions to some of the shortcomings in current approaches to software modernisation, as observed in chapter 1. Specifically in Chapter 4, the thesis proposed a unified approach, REMOST (Re-Engineering through MOdel conStruction and Transformation), in the context of MDA, which integrates all technical supports into a systematic method for software modernisation. Concretely, the original contributions of this thesis are as follows:

C1: In Chapter 5, Wide Spectrum Language (WSL) is extended with a spectrum of modelling languages, called WSL-based Modelling Language (WML), which includes Common Modelling Language (CML), Architecture Description Language (ADL) and Domain Specific Modelling Language (DSML). The work on WSL extensions is done in the wider context of MDA that is a natural continuation of the software transformation framework.

C2: In Chapter 6, model transformation language, *Meta*WML, is defined based on *Meta*WSL. A set of query facilities, action primitives and metric functions are established to unify program and model transformation in a seamless way.

C3: In Chapter 7, a great deal of effort is devoted to unifying WML and UML to bridge legacy systems with MDA.

C4: In Chapter 7, a framework for model construction and transformation based on WML and *Meta*WML is presented. Methodology of model abstraction and refactoring is designed with modern software notions (e.g. design patterns and aspects) that allow for the architecture centred model identification and transformation.

C5: In Chapter 8, a set of toolsets are developed to demonstrate the effectiveness of the proposed approach by re-engineering demonstrator applications into modern paradigm.

10.2.1 Research Questions Revisited

To highlight the significance of contributions made by this thesis, the work is evaluated by answering the proposed research questions. The overall research question presented in Chapter 1 was:

> **How can software models be used to modernise the legacy systems through model construction and transformation?**

The question has been answered in general by proposing an approach called REMOST. In order to be able to answer this question in detail, a set of research questions were defined.

RQ1: What is the model driven software modernisation?

Simply speaking, model driven software modernisation is to use models and modelling techniques as the main artefacts and activities in software modernistion.

RQ2: How can the models be extracted from legacy systems?

In Chapter 7, an algorithm was introduced to extract models from legacy systems.

- *What type of models is required to re-engineer the legacy system?*

In Chapter 4, feature model, architecture model, design model and source code are chosen as a set of models which can specify the legacy system from several viewpoints.

- *How to specify the modelling languages?*

Chapter 5 shows how to specify the modelling languages.

- *How to link the source code with models?*

Chapter 7 shows how to extract models from legacy systems and build the links.

RQ3: How model transformation can be implemented?

Chapter 6 specifies the transformation language and Chapter 7 implements model

transformation with transformation language.

- *How to present model transformation and define model transformation rules with model transformation language?*

Chapter 6 defines model transformation language and shows how to present model transformation and define model transformation rules with model transformation language.

- *How to preserve the traceability during the model transformation?*

Chapter 7 shows that traceability is preserved during the model abstraction and refactoring.

- *What is the role of UML in model driven software modernisation?*

UML is a well-established industry standard, well-known by many industry developers and with good tool support. The system, as represented by UML diagrams, has a better chance of being properly understood by industry developers.

RQ4: How can tool support be provided for the proposed approach?

Chapter 8 (Tool Support) and Chaper 9 (Case Study) show how the tools support the REMOST approach.

10.2.2 Research Hypothesis Revisited

The main hypothesis underlying the present thesis is that software modelling techniques are useful means to manage software modernisation in large complex software systems in a cost efficient manner. The REMOST approach and case studies have shown that this hypothesis is the case.

H1: None of current techniques alone is sufficient to prevent the legacy crisis, but integration of various techniques can reduce the overall effort required to maintain the ever-increasing amount of legacy software code.

Due to this hypothesis, an integration of various techniques in this thesis is used to construct the REMOST approach.

H2: The proposed approach adopts a component-based perspective, assuming that legacy systems are componentised through modular or aspectual decomposition, and supporting gradual migration approach.

This hypothesis is the foundation of algorithm design.

H3: Only parts of the re-engineering process can be automated.

This hypothesis shows that human intervention is necessary.

H4: Not all parts of a legacy system are of equal value and only small parts of a system are representing real business rules and process logic. In MDA context, the technical infrastructure code needs only be re-engineered into models in coarse granularity.

This hypothesis makes software modernisation focus on extraction of Platform Independent Model (PIM).

10.2.3 Success Criteria Revisited

In Chapter 1, a set of criteria are proposed to judge the success of the approach described in this thesis. In this section, detailed analysis of the proposed approach is presented based on these criteria.

What kind of legacy systems can the approach deal with?

The REMOST approach is suited to modernise the legacy systems, where the source code can be obtained. Case studies have shown that the REMOST approach can extract models of various legacy systems from their source code no matter whether they had been modified or were well structured. The transformation rules are based on WSL/WML and hence are language independent. However, the more structured a legacy system is, the easier the extraction process can be.

Can this approach support the modern paradigms like multi-agent systems or Web services?

The answer is yes. WML and *Meta*WML are designed for modern paradigms and provide a lot of supporting functionality. The REMOST approach has flexibility and extensibility as the main target of building a unified solution to system modernisation. Actually, model driven approach is the requirement of modern paradigms, which are always looking for ways to improve software development productivity as well as the quality and longevity of the software.

Are the extracted models consistent to the original design? Are the extracted models unambiguous and easy to understand? Is it reliable to perform forward engineering on the base of the extracted models?

The answer is positive. Although information will be lost during the abstraction, the remained information is structural information which preserves the traceability and new design can be generated for each change in code. The extracted specification is also easy to understand by unifying WML and UML. The case studies conducted also show positive evidence to this conclusion.

Is the approach feasible for realisation? For example, is it possible to build a practical tool to demonstrate the approach? Is the approach capable for industrial-scaled systems?

Actually, quite a lot attention was paid to the practical part of the approach during development. The approach adopts systematic stepwise model construction and transformation (abstraction and refactoring), and a semi-automatic tool has been built, which much improves the efficiency of re-engineering process, together with the problem size. The case studies confirm that the approach is scalable and capable for industrial-scaled systems and efficient enough for real practice. But still, more large industrial-scaled systems should be used for further testing.

10.3 Limitations

Besides the success criteria mentioned above, it is believed that the proposed approach had additional successes as well as some challenges.

Higher level models cannot always be extracted from source code completely.

Some models can not satisfactorily be extracted from source code. These models are highly dependent on user interaction to identify external actors and their roles or are behaviour models of highly reactive systems with many external events. Meanwhile, the use of static information is very useful in understanding the design and structure of a system, but reveals nothing about the behaviour of the system at run-time. Although business processes can be modelled, the external actors to this system need to be identified and this identification is impossible relying just on source code. A developer/user community must be available in order to identify these external actors and the roles that they play in regards to the system or these actors with their roles must be available in up-to-date documentation.

Most of transformation rules in REMOST can not be proved but need to be validated by human beings.

Although transformation techniques can ensure consistency and provide a reliable linkage between the various stages in system development, in order to jump from one level up to another abstract level in the process of reverse engineering, one has to throw away some information. No method can guarantee that such a throwing away of information is appropriate and accurate. This implies that the abstraction is creative work. In order to achieve correct and practical abstraction, human interaction and knowledge base for transformation engine are necessary.

10.4 Conclusion and Future Directions

Based on the discussions in former sections, it can be concluded that REMOST approach is a powerful and systematic means for re-engineering while the prototype tool

environment and case studies show the success of the approach. Because of the very high costs of software modernisation, even a 5% reduction in costs from such a method would be very significant. The use of REMOST will speed up the software modernisation process thus allowing software development groups to be more responsive to their customers' needs, which will at least alleviate the legacy crisis.

Through developing the REMOST approach (including the toolset, FIP, developed in the project), the following lessons are learnt:

- **Models** - WML provides a spectrum of models for re-engineered system, from concrete code to design model, software architecture and domain model. These models in different abstraction levels are integrated and cooperated in a uniform manner.

- **Tools** - The availability of tools greatly facilitated the re-engineering process. A methodology is a systematic approach to solving a problem. The proposed approach is a methodology, which prescribes a set of steps and work products as well as rules to guide the production and analysis of reverse engineering process. The functionality of a tool is not only driven by the task that the tool is designed to perform but also by the methods used to accomplish the task. Automated support for a methodology can aid its use and effectiveness.

The research presented in this thesis is not the terminus. There are a number of areas of future work that can be pursued based on the present work:

- Although domain features and domain knowledge are considered carefully in REMOST method, more profound study of specific domain knowledge could help improve the automation of the tool further. Extraction of domain knowledge is outside the scope of this thesis but also a very relevant area.

- Model Compiler is the core of automation of abstraction and transformation rules. The current research has implemented a parser to extract information from current program and models. The design and implementation of a complete compiler is beyond the current work but should be a priority for future research.

- In order to better determine the industrial effectiveness of the proposed approach and its tools, more case studies should be conducted. There is not enough experiment done due to limited time of this research. In future work, this should be another priority in industrial areas.

References

[1] ACME, "Acme Project", http://www.cs.cmu.edu/~acme/, Acme Team (ABLE group at Carnegie Mellon University and Dave Wile at USC's Information Sciences Institute).

[2] R. S. Arnold, *A Road Map Guide to Software Re-engineering*, IEEE Computer Society Press, 1992.

[3] C. Atkinson and T. Kuehne, "Aspect-Oriented Development with Stratified Frameworks", *IEEE Software*, vol. 20(1), Jan./Feb. 2003, pp. 81-89.

[4] ATLAS, "ATL: Atlas Transformation Language", ATL User Manual (Version 0.7), ATLAS Group, LINA & INRIA, Nantes, Feb. 2006.

[5] T. Baar and J. Whittle, "On the Usage of Concrete Syntax in Model Transformation Rules", *6th International Andrei Ershov Memorial Conference Perspectives of System Informatics*, 2006

[6] F. Bachmann, L. Bass, G. Chastek, et al., "The Architecture Based Design Method", Technical Report CMU/SEI-2000-TR-001, Software Engineering Institute, Carnegie Mellon University, 2000.

[7] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[8] P. Baumann, J. Fassler, M. Kiser, et al., "Semantics-based Reverse Engineering", Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland, 1994.

[9] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *IEEE Computer*, Jul. 1989, pp. 36-49.

[10] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Addison-Wesley, 2000.

[11] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software

Engineering", *IEEE Computer*, vol. 20(4), Apr. 1987, pp. 10-19.

[12] K. Brown, "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk", Master Thesis, University of Illinois at Urbana-Champaign, 1997.

[13] T. Bull, "An Introduction to the WSL Program Transformer", Technical Report, University of Durham, England, 1995.

[14] F. Buschmann, R. Meunier, H. Rohnert, et al., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons Ltd., 1996.

[15] F. Chen, M. Ladkau, S. Natelberg, et al., "Extracting UML Diagrams from a Legacy System", Technical Report, STRL, Department of Computer Science, De Montfort University, UK, Sep. 2005.

[16] F. Chen, S. Li, H. Yang, et al., "Feature Analysis for Service-Oriented Reengineering", *12th IEEE ASIA-PACIFIC Software Engineering Conference (APSEC'05)*, Taipei, Taiwan, Dec. 2005

[17] F. Chen, H. Yang, H. Guo, et al., "Aspect-Oriented Programming based Software Evolution with Microsoft .NET." *21st IEEE International Conference on Software Maintenance (4 pages poster paper)*, Budapest, Hungary, Sep. 2005

[18] F. Chen, H. Yang, H. Guo, et al., "Agentification for Web Service", *28th IEEE International Computer Software and Application Conference (COMPSAC'04)*, Hong Kong, Sep. 2004, pp. 514-519.

[19] F. Chen, H. Yang and B. Qiao, "A Formal Model Driven Approach to Dependable Software Evolution", *30th IEEE International Computer Software and Application Conference (COMPSAC'06)*, Chicago, USA, Sep. 2006

[20] K. Chen, J. Sztipanovits and S. Neema, "Toward a Semantic Anchoring Infrastructure for Domain-specific Modeling Languages", *5th ACM International Conference on Embedded Software*, 2005, pp. 35-43.

[21] Z. Chen, H. Zedan, A. Cau, et al., "A Wide-Spectrum Language for Object-Based Development of Real-time Systems", *International Journal of Information Sciences*, vol. 118, 1999, pp. 15-35.

[22] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: a Taxonomy", *IEEE Software*, vol. 7(1), Jan. 1990, pp. 13-17.

[23] M. O. Cinneide, "Automated Refactoring to Introduce Design Patterns", *IEEE International Conference on Software Engineering*, 2000, pp. 722-724.

[24] F. Cioch, M. Palazzolo and S. Lohrer, "A Documentation Suite for Maintenance Programmers", *1996 International Conference on Software Maintenance (ICSM'96)*, Nov. 1996, pp. 286-295.

[25] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys*, vol. 28(4), Sep. 1996, pp. 626-643.

[26] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", *23rd IEEE International Conference on Software Engineering*, Toronto, Canada, May 2001

[27] Y. Coady, G. Kiczales, M. Feeley, et al., "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code", *9th ACM SIGSOFT International Symposium*, Vienna, Austria, Sep. 2001

[28] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches", *Workshop on Generative Techniques in the Context of Model-Driven Architectures (OOPSLA '03)*, 2003

[29] K. Czarnecki and U. W. Eisenecker, *Generative Programming*, Addison Wesley, 2000.

[30] A. Deursen and T. Kuipers, "Building Documentation Generators", *IEEE International Conference on Software Maintenance (ICSM'99)*, Sep. 1999, pp. 40-49.

[31] A. V. Deursen and P. Klint, "Domain-Specific Language Design Requires Feature Descriptions", *Journal of Computing and Information Technology*, vol. 10(1), 2002, pp. 1-17.

[32] P. Devanbu and E. Wohlstadter, "Evolution in Distributed Heterogeneous Systems", *RAM-SE'04 ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, Oslo, Norway, Jun. 2004

[33] J. Dong, S. Yang and K. Zhang, "A Model Transformation Approach for Design Pattern Evolutions", *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, 2006, pp. 80-92.

[34] A. Egyed, "Automated Abstraction of Class Diagrams", *ACM Transactions on Software Engineering and Methodology*, vol. 11(4), Oct. 2002, pp. 449-491.

[35] T. Eisenbarth, R. Koschke and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29(3), Mar. 2003, pp. 210-224.

[36] D. W. Embley, B. D. Kurtz and S. N. Woodfield, *Object-Oriented Systems Analysis: a Model-Driven Approach*, Prentice Hall, 1992.

[37] ExpressLogic, "ThreadX User Guide", Express Logic, Inc., http://www.expresslogic.com.

[38] J. Favre, "Foundations of Model (Driven) (Reverse) Engineering: Models", *Int. workshop Dagsthul*, 2004

[39] J. Favre, "Towards a Basic Theory to Model Model Driven Engineering", *3rd Workshop on Software Model Engineering (WiSME'04)*, Lisboa, Portugal, Oct. 2004

[40] N. E. Fenton and S. L. Pfleeger, *Software Metrics - A Rigorous Approach, Second Edition*, International Thomson Computer Press, 1996.

[41] R. E. Filman and D. P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, Oct. 2000

[42] N. Fletton and M. Munro, "Redocumenting Software Systems Using Hypertext Technology", *1988 Conference on Software Maintenance (CSM'88)*, Oct. 1988, pp. 54-59.

[43] B. Flyvbjerg, "Five Misunderstandings About Case Study Research", *Qualitative Inquiry*, vol. 12(2), Apr. 2006, pp. 219-245.

[44] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*, Addison Wesley, Sep. 2003.

[45] P. Fradet and M. Sudholt, "AOP: towards a Generic Framework using Program Transformation and Analysis", *International Workshop on Aspect-Oriented Programming at ECOOP'98*, Brussels, Belgium, Jul. 1998

[46] D. S. Frankel, *Model Driven Architecture - Applying MDA to Enterprise Computing*, Wiley Publishing, Inc., 2003.

[47] E. Gamma, R. Helm, R. Johnson, et al., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[48] D. Garlan, R. T. Monroe and D. Wile, "Acme: Architectural Description of Component-Based Systems", in *Foundations of Component-Based Systems*, Cambridge University Press, 2000.

[49] D. Garlan and M. Shaw, "An Introduction to Software Architecture", Technique Report, CMU/SEI-94-TR-21, Carnegie Mellon University, 1994.

[50] E. Garson, "Aspect-Oriented Programming in C#/.NET", *Visual Systems Journal (VSJ)*, Feb. 2004.

[51] M. C. W. Geilen, J. P. M. Voeten, P. H. A. van-der-Putten, et al., "Object-Oriented Modelling and Specification Using SHE", *Journal of Computer Languages*, vol. 27 (3), Dec. 2001, pp. 19-38.

[52] R. Gerlich and U. Denskat, "A Cost Estimation Model for Maintenance and High Reuse", *ESCOM 1994*, Ivrea, Italy, 1994

[53] K. Gowthaman, K. Mustafa and R. A. Khan, "Reengineering Legacy Source Code to Model Driven Architecture", *4th Annual ACIS International Conference on Computer and Information Science (ICIS'05)*, 2005, pp. 262-267.

[54] M. Griss, "Implementing Product-Line Features with Component Reuse", *International Conference on Software Reuse (ICSR'00)*, Vienna, Austria, Jun. 2000

[55] P. Grubb and A. Takang, *Software Maintenance: Concepts and Practice, 2nd Edition*, World Scientific Publishing Co., Singapore, 2003.

[56] G. Guo, J. Atlee and R. Kazman, "A Software Architecture Reconstruction Method", *1st Working IFIP Conference on Software Architecture (WICSA)*, San

Antonio, Texas, Feb. 1999, pp. 225-243.

[57] H. Guo, C. Guo, F. Chen, et al., "Wrapping Client-Server Application to Web Services for Internet Computing", *6th IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05)*, Dalian, China, Dec. 2005

[58] G. T. Heineman and W. T. Council, *Component-Based Software Engineering: Putting the Pieces Together*, Addsion-Wesley, 2001.

[59] M. Herr, U. Bath and A. Koschel, "Implementation of a Service Oriented Architecture at Deutsche Post MAIL", *Lecture Notes in Computer Science 3250, Springer, ECOWS 2004*, Erfurt, Germany, Sep. 2004, pp. 227-238.

[60] H. M. Hess, "Aligning Technology and Business: Applying Patterns for Legacy Transformation", *IBM Systems Journal*, vol. 44(1), 2005, pp. 25-45.

[61] W. Heuvel, "Matching and Adaptation: Core Techniques for MDA-(ADM)-Driven Integration of New Business Applications with Wrapped Legacy Systems", *IEEE MELS Workshop*, 2004

[62] C. Hofmeister, R. Nord and D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.

[63] M. N. Huhns, "Software Development with Objects, Agents, and Services", *3rd International Workshop on Agent-Oriented Methodologies (Keynotes)*, Vancouver, Canada, Oct. 2004

[64] IBM, http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/.

[65] IEEE, "IEEE Standard Collection: Software Engineering", IEEE Inc., New York, 1997.

[66] IEEE, "The Open Group Base Specifications Issue 6. IEEE Std 1003.1-2001", The IEEE and The Open Group, 2001.

[67] IO, "Application Modernization and Legacy-to-SOA", Interactive Objects, http://www.interactive-objects.com/solutions/application-modernization/.

[68] IOSoftware, http://www.io-software.com, Interactive Objects Software GmbH.

[69] T. Ishio, S. Kusumoto and K. Inoue, "Program Slicing Tool for Effective Software Evolution using Aspect-Oriented Technique", *6th IEEE International Workshop on Principles of Software Evolution (IWPSE'03)*, Helsinki, Finland, Sep. 2003

[70] ITU, *Specification and Description Language (SDL)*, ITU-T Recommendation Z.100, 1999.

[71] JavaCC, "Java Compiler Compiler", https://javacc.dev.java.net/.

[72] N. R. Jennings, "On Agent-Based Software Engineering", *Artificial Intelligence*, vol. 117, 2000, pp. 277-296.

[73] K. Kang, S. Cohen, J. Hess, et al., "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[74] K. C. Kang, S. Kim, J. Lee, et al., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, vol. 5, 1998, pp. 143-168.

[75] R. Kazman, M. Klein and P. Clements, "ATAM: Method for Architecture Evaluation", Technical Report CMU/SEI-2000-TR-004, ADA382629, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 2000.

[76] R. Kazman, L. O'Brien and C. Verhoef, "Architecture Reconstruction Guidelines, 3rd Edition", Technical Report CMU/SEI-2002-TR-034, ESC-TR-2002-034, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, Nov. 2003.

[77] S. Kent, "Model Driven Engineering", *3rd International Conference on Integrated Formal Methods (IFM'02), LNCS 2335*, 2002, pp. 286-298.

[78] J. Kerievsky, *Refactoring to Patterns*, Addison Wesley, 2005.

[79] G. Kiczales, E. Hilsdale, J. Hugunin, et al., "An Overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP'01)*, Budapest, Hungary, Jun. 2001

[80] G. Kiczales, J. Lamping, A. Mendhekar, et al., "Aspect-Oriented Programming",

European Conference on Object-Oriented Programming (ECOOP'97), Finland, Jun. 1997, pp. 220-242.

[81] A. Kleppe, J. Warmer and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison Wesley, 2003.

[82] R. L. Krikhaar, "Reverse Architecting Approach for Complex Systems", *International Conference on Software Maintenance*, 1997, pp. 4-11.

[83] R. L. Krikhaar, "Software Architecture Reconstruction", PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1999.

[84] P. B. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, vol. 12(6), Nov. 1995, pp. 42-50.

[85] I. Kurtev, J. Bezivin and M. Aksit, "Technical Spaces: an Initial Appraisal", *Confederated International Conferences (CoopIS, DOA'02), Industrial Track*, Irvine, 2002

[86] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, USA, 2003.

[87] M. Ladkau, F. Chen, S. Natelberg, et al., "FermaT Transformation Engine Tutorial", Technical Report, STRL, Department of Computer Science, De Montfort University, UK, Nov. 2006.

[88] K. Lee, K. C. Kang, W. Chae, et al., "Feature-based Approach to Object-Oriented Engineering of Applications for Reuse", *Journal of Software Practice and Experience*, vol. 30, 2000, pp. 1025-1046.

[89] M. M. Lehman, "Laws of Software Evolution Revisited", *LNCS 1149*, 1997, pp. 108-124.

[90] M. M. Lehman and J. F. Ramil, "Towards a Theory of Software Evolution and Its Practical Impact", *International Symposium on the Principles of Software Evolution, Invited Talk*, Washington, USA, 2000, pp. 2-11.

[91] M. M. Lehman and J. F. Ramil, "Software Evolution and Software Evolution Processes", *Annals of Software Engineering*, vol. 14(1-4), 2002, pp. 275-309.

[92] M. M. Lehman and J. F. Ramil, "Software Evolution in the Age of

Component-Based Software Engineering", *IEE Proceedings Software*, Dec. 2000, pp. 249-255.

[93] S. Li, H. Yang and H. Zhou, "Building a Dependable Enterprise Service Assembly Line (ESAL) for Legacy Application Integration", *2004 International Conference on Cyberworlds (CW'04)*, Tokyo, Japan, Nov. 2004, pp. 170-175.

[94] T. Li, H. Yang, B. Xu, et al., "An Approach to Transforming Parallel Function Specification into Java Program Framework", *International Conference on Software Engineering Research and Practice (SERP'05), Volume 2*, Las Vegas, USA, Jun. 2005, pp. 517-523.

[95] Y. Li, "Automating Domain Knowledge Recovery from Legacy Software Code", PhD Thesis, De Montfort University, England, 2002.

[96] Y. Li, H. Yang and W. C. Chu, "A Concept-Oriented Belief Revision Approach to Domain Knowledge Recovery from Source Code", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13(1), 2001, pp. 31-52.

[97] X. Liu, "Abstraction: A Notation for Reverse Engineering", PhD Thesis, De Montfort University, England, 1999.

[98] X. Liu, H. Yang, H. Zedan, et al., "Speed and Scale Up Software Reengineering with Abstraction Patterns and Rules", *International Symposium on Software Evolution*, Japan, 2000

[99] S. Lu, W. A. Halang and R. Gumzej, "Towards Platform Independent Models of Real Time Operating Systems", *2nd IEEE International Conference on Industrial Informatics*, 2004, pp. 249 - 254.

[100] A. MacDonald, D. Russell and B. Atchison, "Model-driven Development within a Legacy System: An Industry Experience Report", *2005 Australian Software Engineering Conference (ASWEC'05)*, 2005, pp. 14-22.

[101] M. W. Maier, D. Emery and R. Hilliard, "Software Achitecture: Introducing IEEE Standard 1471", *IEEE Computer*, vol. 34(4), 2001, pp. 107-109.

[102] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven*, Addison Wesley, 2002.

[103] S. J. Mellor, K. Scott, A. Uhl, et al., *MDA Distilled: Principle of Model Driven Architecture*, Addison Wesley, 2004.

[104] A. Metha and G. T. Heineman, "Evolving Legacy System Features into Fine-Grained Components", *International Conference of Software Engineering (ICSE'02)*, Orlando, USA, May 2002

[105] R. C. Millham, "Evolution of Batch-Oriented COBOL Systems into Object-Oriented Systems through Unified Modelling Language", PhD Thesis, De Montfort University, England, 2006.

[106] R. C. Millham, J. Pu and H. Yang, "TAGDUR: A Tool for Producing UML Sequence, Deployment, and Component Diagrams Through Reengineering of Legacy Systems", *IASTED Conference on Software Engineering*, Innsbruck, Austria 2004

[107] R. C. Millham and H. Yang, "TAGDUR: A Tool for Producing UML Diagrams Through Reengineering of Legacy Systems", *IASTED Conference on Software Engineering*, Marina del Ray, USA, 2003

[108] D. Millicev, "Domain Mapping Using Extended UML Object Diagrams", *IEEE Software*, pp. 90-97.

[109] B. Moszkowski, *Executing Temporal Logic Programs*, Cambridge University Press, 1986.

[110] H. Muller, M. Orgun, S. Tiley, et al., "A Reverse Engineering Approach to Subsystem Structure Identification", *Software Maintenance: Research and Practise, No. 5*, 1993, pp. 181-204.

[111] P.-A. Muller, *Instant UML*, Wrox, Birmingham, UK, 2000.

[112] T. Myers, *Equations, Models, and Programs: a Mathematical Introduction to Computer*, Prentice Hall, 1988.

[113] N. Noda and T. Kishi, "On Aspect-Orientation in Distributed Real-Time Dependable Systems", *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02)*, San Diego, USA, Jan. 2002, pp. 261-267.

[114] OMG, "XML Metadata Interchange (XMI), v2.0 specification", omg/formal/03-05-01, 2003.

[115] OMG, "MOF 2.0/XMI Mapping Specification, v2.1", omg/formal/03-05-01, 2005.

[116] OMG, "Architecture Driven Modernization Roadmap", Technical Report, Draft #1, ADM Task Force, OMG, http://adm.omg.org/ADMTF Roadmap.pdf, 2006.

[117] OMG, "Object Constraint Language, v2.0", omg/formal/06-05-01, 2006.

[118] OMG, "Meta Object Facility (MOF) Specification v1.4", Apr. 2002.

[119] OMG, "MOF 2.0 Query/ Views/ Transformations RFP, ad/2002-04-10", Apr. 2002.

[120] OMG, "OMG Document: ad/98-07-07 Version 1.2 /T-UOL-19980707", Recerca Informatica, Daimler-Benz Research and Technology, Jul. 1998.

[121] OMG, "MDA Guide Version 1.0.1 ", omg/2003-06-01, Jun. 2003.

[122] OMG, "UML 2.0 Superstructure Specification", Document: ptc/04-10-02 (convenience document), Oct. 2004.

[123] R. F. Paige, J. S. Ostroff and P. J. Brooke, "Principles for Modeling Language Design", *Information & Software Technology*, vol. 42(10), 2000, pp. 665-675.

[124] D. L. Parnas, "Software Aging", *16th International Conference on Software Engineering*, Sorrento, Italy, 1994, pp. 279-287.

[125] D. L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transaction on Software Engineering*, vol. 5(2), March 1979, pp. 128-138.

[126] I. Pashov, "Feature-based Methodology for Supporting Architecture Refactoring and Maintenance of Long-life Software Systems", PhD Thesis, Technical University Ilmenau, 2004.

[127] S. L. Pfleeger, *Software Engineering; Theory and Practice (2nd Edition)*, Pearson Education, USA, 1998.

[128] C. Pons and G. Baum, "Formal Foundations of Object-Oriented Modeling Notations", *3rd IEEE International Conference on Formal Engineering Methods*

(ICFEM'00), York, UK, 2000, pp. 101-110.

[129] A. Popovici, T. Gross and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming", *1st International Conference on Aspect-Oriented Software Development (AOSD), Sponsored by the University of Twente and IBM Research etc.*, Netherlands, Apr. 2002

[130] R. S. Pressman, *Software Engineering: A Practitioner's Approach (5th Ed)*, McGraw-Hill, 2001.

[131] B. Qiao, "Evolution of WEB-based Systems in Model Driven Architecture", PhD Thesis, De Montfort University, England, 2005.

[132] B. Qiao, H. Yang, W. C. Chu, et al., "Bridging Legacy Systems to Model Driven Architecture", *27th IEEE International Computer Software and Application Conference (COMPSAC'03)*, Washington, USA, 2003, pp. 304-309.

[133] V. Rajlich, "Incremental redocumentation Using the Web", *IEEE Software*, vol. 17(5), Sep.-Oct. 2000, pp. 102-106.

[134] T. Reus, H. Geers and A. v. Deursen, "Harvesting Software Systems for MDA-Based Reengineering", in *Model Driven Architecture – Foundations and Applications*, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 213-225.

[135] M. Riebisch, "Towards a More Precise Definition of Feature Models", *Modelling Variability for Object-Oriented Product Lines ECOOP Workshop*, Germany, Jul. 2003, pp. 64-76.

[136] M. Riebisch, "Supporting Evolutionary Development by Feature Models and Traceability Links", *11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'04)*, Brno, Czech Republic, May 2004

[137] S. Robitaille, R. Schauer and R. Keller, "Bridging Program Comprehension Tools by Design Navigation", *International Conference on Software Maintenance (ICSM'00)*, Oct. 2000, pp. 22-32.

[138] RTLinux, "RTLinux V3.0 Source Code", ftp://ftp.rtlinux.com/pub/rtlinux/v3/.

[139] M. Salah and S. Mancoridis, "A Hierarchy of Dynamic Software Views: from Object-Interactions to Feature-Interactions", *International Conference of Software Maintenance (ICSM'04)*, Chicago, USA, Sep. 2004

[140] F. Schull, W. Melo and W. Basili, "An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems", UMIACS-TR-96-10, University of Maryland, 1996.

[141] W. Schult and A. Polze, "Aspect-Oriented Programming with C# and .NET", *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, Washington USA, 2002, pp. 241-249.

[142] R. C. Seacord, D. Plakosh and G. A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*, Addison Wesley, 2003.

[143] E. Seidewitz, "What Models Mean", *IEEE Software*, Sep. 2003.

[144] B. Selic, G. Gullekson and P. T. Ward, *Real-Time Object-Oriented Modeling*, Wiley & Sons, New York, 1994.

[145] S. Sendall, R. Hauser, J. Koehler, et al., "Understanding Model Transformation by Classification and Formalization", *Workshop on Software Transformation Systems (part of 3rd International Conference on Generative Programming and Component Engineering)*, Vancouver, Canada, Oct. 2004

[146] M. Shaw and D. Garlan, *Software Architecture-Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[147] I. Sommerville, *Software Engineering, 5th Edition*, Addison Wesley, 1995.

[148] S. Souza, N. Anquetil and K. Oliveira, "A Study of the Documentation Essential to Software Maintenance", *23rd Annual International Conference on Design of Communication (SIGDOC'05)*, Sep. 2005, pp. 68-75.

[149] C. Stoermer, A. Rowe, L. O'Brien, et al., "Model-centric Software Architecture Reconstruction", *Software: Practice and Experience*, 2005, pp. 333-363.

[150] M. D. Storey, F. D. Fracchia and H. A. Muller, "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization",

5th Workshop on Program Comprehension (WPC'97), Dearborn, USA, 1997, pp. 17-28.

[151] E. Stroulia, M. El-Ramly and P. Sorenson, "From Legacy to Web through Interaction Modeling", *International Conference on Software Maintenance (ICSM'02)*, Montrzal, Canada, Oct. 2002, pp. 320-329.

[152] Sun, "MetaData Repository", http://mdr.netbeans.org/.

[153] J. Sztipanovits and G. Karsai, "Model-Integrated Computing", *IEEE Computer*, Apr. 1997, pp. 110-112.

[154] L. Tahvildari, "Quality-Driven Object-Oriented Re-engineering Framework", PhD Thesis, University of Waterloo, Canada, 2003.

[155] S. Tilley, H. Muller and M. Orgun, "Documenting Software Systems with Views", *10th International Conference on Systems Documentation (SIGDOC'92)*, 1992, pp. 211-219.

[156] C. R. Turner, A. Fuggetta, L. Lavazza, et al., "A Conceptual Basis for Feature Engineering", *Journal of System and Software*, vol. 49(1), Dec. 1999, pp. 3-15.

[157] P. H. A. van-der-Putten and J. P. M. Voeten, "Specification of Reactive Hardware/Software Systems - The Method Software/Hardware Engineering (SHE)", PhD Thesis, Technische Universiteit Eindhoven, The Netherlands, 1997.

[158] S. Vinoski, "Web Services Interaction Models, Part 1: Current Practice", *IEEE Internet Computing*, vol. 6(3), May-Jun. 2002, pp. 89-91.

[159] D. Vollmann, "Visibility of Join-Points in AOP and Implementation Languages", *2nd Workshop on Aspect-Oriented Software Development*, Bonn, Germany, Feb. 2002, pp. 65-69.

[160] W3C, "Web Services Architecture", http://www.w3.org/TR/2004/NOTE-ws-arch-20040211, W3C Working Group Feb. 2004.

[161] W3C, "Web Services Architecture Requirements", http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211, W3C Working Group,

Feb. 2004.

[162] Y. Wang, F. Chen, L. Xu, et al., "An Implementation of a Multi-Interface Virtual Real-Time Operating System on Windows Platform", *Journal of Dalian University of Technology*, vol. Suppl., 43(S1), Oct. 2003, pp. 100-102 (in Chinese).

[163] M. Ward, "Proving Program Refinements and Transformations", PhD Thesis, Oxford University, England, 1989.

[164] M. Ward, "Language Oriented Programming", *Software - Concepts and Tools*, vol. 15(4), 1994, pp. 147-161.

[165] M. Ward, "Program Analysis by Formal Transformation", *Computer Journal*, vol. 39(7), 1996, pp. 598–618.

[166] M. Ward, "The FermaT Assembler Re-engineering Workbench", *IEEE International Conference on Software Maintenance (ICSM'01)*, 2001, pp. 659-662.

[167] M. Ward, "Program Slicing via FermaT Transformations", *26th IEEE Annual International Computer Software and Applications Conference (COMPSAC'02)*, Oxford, England, Aug. 2002

[168] M. Ward and H. Zedan, "MetaWSL and Meta-Transformations in the FermaT Transformation System", *29th IEEE Annual International Computer Software and Applications Conference (COMPSAC'05)*, 2005, pp. 233-238.

[169] M. Ward and H. Zedan, "Slicing as a Program Transformation", *ACM Transactions on Programming Languages and Systems*, vol. 29(2), Mar. 2007.

[170] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley, 2003.

[171] I. Warren, *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*, Springer-Verlag, 1999.

[172] N. Wilde, M. Buckellew, H. Page, et al., "A Comparison of Methods for Locating Features in Legacy Software", *Journal of Systems and Software*, vol. 65(2), Feb. 2003, pp. 105-114.

[173] M. Wimmer and G. Kramler, "Bridging Grammarware and Modelware", in *Lecture Notes in Computer Science: Satellite Events at the MoDELS 2005 Conference*, vol. 3844, 2006, pp. 159-168.

[174] K. Wong, S. Tilley, H. Muller, et al., "Structural Redocumentation: a Case Study", *IEEE Software*, vol. 12(1), Jan. 1995, pp. 46-54.

[175] R. Wuyts, "A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation", PhD Thesis, Vrije University, Brussel, 2001.

[176] H. Yang, "The Supporting Environment for A Reverse Engineering System - The Maintainers Assistant", *IEEE Conference on Software Maintenance 1991*, Sorrento, Italy, Oct. 1991, pp. 13-22.

[177] H. Yang and K. Bennett, "Acquisition of ERA Models from Data Intensive Code", *International Conference on Software Maintenance (ICSM'95)*, Nice, France, 1995, pp. 116-123.

[178] H. Yang, Z. Cui and P. O'Brien, "Extracting Ontologies from Legacy Systems for Understanding and Re-Engineering", *23rd IEEE Annual International Computer Software and Applications Conference (COMPSAC'99)*, Washington, USA, 1999, pp. 21-26.

[179] H. Yang, X. Liu and H. Zedan, "Abstraction: A Key Notion for Reverse Engineering in a System Reengineering Approach", *Journal of Software Maintenance: Research and Practice*, vol. No.12, 2000, pp. 197-228.

[180] H. Yang and M. Ward, *Successful Evolution of Software Systems*, Artech House, Inc., 2003.

[181] U. Zdun, "Reengineering to the Web: A Reference Architecture", *6th European Conference on Software Maintenance and Reengineering (CSMR'02)*, 2002, pp. 164-176.

[182] H. Zedan, S. Zhou, N. Sampat, et al., "K-Mediator: Towards Evolving Information Systems", *IEEE International Conference on Software Maintenance (ICSM'01)*, 2001, pp. 520-527.

[183] Z. Zhang and H. Yang, "Incubating Services in Legacy Systems for Architectural Migration", *11th IEEE Asia-Pacific Software Engineering Conference (APSEC'04)*, Busan, Korea, Nov. 2004, pp. 196-203.

[184] H. Zhou, "A Notion of a System Pattern (Tri-integration) and its Instantiation in Enterprise Application Development", PhD Thesis, De Montfort University, England, 2004.

[185] Y. Zou, "Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms", PhD Thesis, University of Waterloo, Canada, 2003.

Appendix A: Grammar Summary of WML

The following is a summary of the context-free grammar for WML. The syntax of complete WML includes syntax of WSL, WML (CML, ADL and FDL). Although a program is a text consisting of individual characters, it is often more convenient to regard literals, identifiers, operators, etc. as atomic symbols called tokens. The syntax of tokens themselves is called the microsyntax, or lexicon of the WML.

A.1 Basic Concepts and Definitions

The central concept of CML for modelling entities of the problem domain is the class. A class is a user-defined type, which provides a common description for a set of objects sharing the same properties. The properties of objects include attributes, describing their data resources, and operations, offering the means for manipulating their data resources and providing functional services for other objects. An object is an instance of a class with unique identity.

Definition A.1 (Identifiers): To identify model components, *IDS* is defined as a set of all possible, infinite, non-empty identifiers, where $| IDS | = \infty$. Identifiers are needed to distinguish classes, objects, processes and components.

Definition A.2 (Types Signature): Types signature is a pair $\sum_T = (T, OP)$, where:

- T is a set of type names $T \subseteq IDS$
- OP is a set of operations over types in T

The type system is a kind of approximate semantics. It should be designed to allow as many reasonable models as possible, without generating false alarms, while still catching prior to analysis those errors. Type system includes types of process, class, interface, or the predefined basic types (Boolean, Integer, String, Real, etc.), which also defines the usual operations such as relational and mathematical operations. Collection

types are available for describing collections of values and structured values, which are described by tuple types. All type domains include an undefined value that allows operating with unknown or "null" values, denoted with $\perp$.

Definition A.3 (Classes Signature): Classes signature is a triple $\sum_C = (C,\{A_c\},\{M_c\})$, where:

- C is a set of classes identified with a finite set of names $C \subseteq IDS$
- For each class $c \in C$, set A_c describes the attributes of class c
- For each class $c \in C$, set M_c describes the methods of class c

A class signature specifies the classes that are present in a specification, together with their attributes and operations. A class can introduce a new user-defined type implicitly. It induces an object type $t_c \in T$ having the same name as the class. A value of an object type refers to an object of the corresponding class. The main difference between classes and object types is that the interpretation of the latter includes $\perp$, a special undefined value.

Definition A.4 (Attributes): Let $t \in T$ be a type. The attributes of a class $c \in C$ are defined as a set A_c of signatures (a, t) where the attribute name $a \in IDS$, and a type $t \in T$.

All attributes of a class have distinct names. In particular, an attribute name may not be used again to define another attribute with a different type.

$\forall (a_i,t_i), (a_j, t_j) \in A_c: a_i = a_j \Rightarrow t_i = t_j$.

Attributes with the same name may, however, appear in different classes.

Definition A.5 (Methods): Let $t_1, \ldots, t_m$ and $t_{m+1}, \ldots, t_n$ be types in T. Methods of a class $c \in C$ with type $t_c \in T$ are defined by a set M_c of signatures $m : t_c \times t_1 \times \ldots \times t_m \rightarrow t_{m+1} \times \ldots \times t_n$ with operation symbols $m \in IDS$.

Methods are part of a class definition. They are used to describe behavioural

properties of objects. The effect of a method can be specified in a declarative way with pre- and post-conditions.

For object-oriented modelling it is recommended to avoid defining attributes that are references to other objects. They should be modelled as relationships instead. Relationships, also called associations or conceptual relations, exist when objects have some form of dependency.

Definition A.6 (Classes Generalisation Hierarchy): A generalisation hierarchy $\prec$ is a partial order on the set of classes C. Pairs in $\prec$ describe a generalisation relationship between two classes. For classes $c_2, c_1 \in C$, $c_2 \prec c_1$ introduces a class named c_2 that is a subclass of c_1.

A class that is declared independently of any other is a top-level class. The extensions of a class are mutually disjoint, as are top-level class. The effect of a collection of class declarations, some top-level, and some as extensions, is thus to introduce a classification hierarchy, which gives a primary classification to all objects. The superclass defines the attributes and methods that are necessary for its instances. The subclass adds additional attributes and methods. A subclass has at least the properties that its superclass has.

Definition A.7 (Associations): The set of associations AS is defined by a set of pair $<ae_1, ae_2>$, where $ae = (rn, c, m, t, nav)$, so called association ends, is defined as a tuple, consisting of:

- a role name $rn \in IDS$,
- a class, $c \in C$,
- a multiplicity $m \in (N_0 \cup \infty) \times (N_0 \cup \infty)$,
- an aggregation type $t \in \{none, aggregate, composite\}$, and
- a Boolean navigability property *nav*

A class signature specifies the classes that are present in a specification, while

associations describe structural relationships between classes. Generally, classes can participate in any number of associations, and associations can connect two or more classes. Since for many problems the use of binary associations is often sufficient, in this research, only binary relationship is expressed by associations.

A self-association (or recursive association) is a binary association where both ends of the association are attached to the same class *c*. Classes can appear more than once in an association each time playing a different role. Therefore each class participating in an association need be assigned a unique role name. Roles are particularly needed when relations are specified between objects of the same class, or for distinguishing two relationships between the same pair of classes. A role name of a class can also be used to determine the navigation path.

Definition A.8 (Objects): The set of objects of a class is defined by $O(c^*) = \{o_1^c, ..., o_n^c\}$, where an object identifier $o_i^c \in IDS \wedge c^*, c \in C \wedge c^* \prec c$.

A class definition serves as a template for all objects of that class. The creation or derivation of an object from a class is called instantiation. An object is an instance from a class. An object is a model of a concept, or of an entity from a problem domain. It is characterised by its externally observable behaviour and its internal state. Objects are the fundamental unit of granularity. An object class definition defines the attributes of its objects. By the mechanism of instantiation, all individual objects of a class have the same set of attributes. However, an attribute may have different values in different objects from that class. The individual objects of the same class can do different things in different states.

Definition A.9 (Links): For each association $as \in AS$, links are interpreted as the Cartesian product of the sets of object identifiers of the participating classes: $L(as) = O(c_1) \times O(c_2)$.

A concrete relationship between a pair of objects is called a link. An association describes possible connections between objects of the classes participating in the association. The interpretation of an association is a relation describing the set of all possible links between objects of the associated classes and their children. Candidate

relationships can be found by looking at all kinds of dependencies of classes.

CML comprises two main hierarchical levels: the process model and the object/procedure model. The process model level indicates the relations between sub-process models and their parent process models. Modern software systems are commonly distributed applications which require modules to be self-contained, autonomous, relatively independent, and weakly coupled entities. Parallelism may be necessary to process a set of these entities concurrently. These entities are called process objects or processes in short. Processes are instantiated from Process Classes. Behaviour of a process is specified in its Class definition. Such a class definition is a formal behaviour description in the form of a Process Class. The idea of making a distinction between process object and common data object in WML is inspired by ROOM [144], SDL [70], and POOSL [51] for concurrency and an event-oriented modelling. A software process model can also be regarded as an object-oriented system.

Definition A.10 (Processes Signature): Processes signature is a triple $\sum_P=(PI, T\_MIN_P, T\_MAX_P)$, where:

- PI is a set of process classes identified with a finite set of names $PI \subseteq IDS$
- T_MIN_P is the number of threads that are attributed to the process when the process is created and which will not be deleted over process lifetime.
- T_MAX_P specifies the upper limit of threads supported in the process.

Concurrency can be described in terms of threads. A thread is defined as a primitive executable processing element within a process. A thread consists of a locus of control and a stack which represents its state and is initially empty.

Definition A.11 (Process Objects): The set of process objects (or processes) of a process class is defined by $P = \{ p_1^{pc}, ..., p_n^{pc} \}$, where an process identifier $p_i^{pc} \in IDS \wedge pc \in PI$.

Each process has a unique process identifier and is comprised of a set of objects and a set of threads. Process is the unit that determine the name space for these objects and

threads. The lifetime of threads and objects is limited to the lifetime of their corresponding process, i.e., the deletion of a process implies the deletion of all threads and objects contained in this process. Process identifiers are used as 'address' of processes for delivering messages/events on channels. The process concept is very useful when the legacy system need to be modernised into service or agent based system.

Definition A.12 (Thread): A thread is a triple *(t_id, t_behav, t_pri)*, where:

- $t\_id \in IDS$ is the thread identifier.
- *t_behav* is the thread behaviour.
- *t_pri* is the thread priority in a process.

Each thread has a unique thread identifier and each thread belongs to exactly one process. In practice, thread pool is frequently being applied in real-time systems where the dynamic creation of threads has to be avoided due to the time-consuming character of such creations. In such a case, both T_MIN_P and T_MAX_P should be set to n where n is the number of threads forming the thread pool.

Definition A.13 (Software Model): A generic software model M is a structure

$M = (\sum_P, \sum_C, A_c, M_c, AS, \prec)$, where:

- $\sum_P$ is a set of process classes,
- $\sum_C$ is a set of general classes,
- A_c is a set of operation signatures for functions mapping an object of class c to an associated attribute value,
- M_c is a set of signatures for user-defined operations of a class c,
- AS is a set of associations,
- $\prec$ is a partial order on $\sum_P$ and $\sum_C$, reflecting the generalisation hierarchy of

classes.

Definition A.14 (System State):

A system state for a model M is a structure $\sigma(M) = (\sigma_P, \sigma_t^{pc}, \sigma_C, \sigma_A, \sigma_{AS})$.

- The finite sets σ_P contain all objects of a process class $p \in PI$ existing in the system state.
- The finite sets σ_t^{pc} contain all threads of a process $pc \in P$ existing in the system state.
- The finite sets σ_C contain all objects of a class $c \in C$ existing in the system state.
- Functions σ_A assign attribute values to each object.
- The finite sets σ_{AS} contain all links connecting objects. A link set must satisfy all multiplicity specifications defined for an association.

Processes, objects, links, attribute values constitute the state of a system at a particular moment in time. A system is in different states as it changes over time. Therefore, a system state is also called a snapshot of a running system.

The definition of architecture is based on the concept of modularity. Modules are units that define interfaces and are used to represent the subsystems. To manage complexity, an abstraction mechanism is required that groups collections of collaborating objects into single entities, namely 'component' that has a well defined interface. Components are the basic building blocks for describing a system that represent the elements of a system responsible for doing the "work" in a system. This is in contrast to connectors which model communication and interaction between components in a system. Intuitively, components correspond to the "boxes" in box and line diagrams [1].

A component can be used to capture any number of abstract characterisations of the

computational aspects of a system. The implementation of component is described by class model and interaction model and interface describes the attributes and behaviours of the components. Components expose their functionality through their ports. A port represents a point of contact between the component and its environment. A component may have several ports corresponding to different interfaces to the component. A port can be used to represent what is traditionally thought of as an interface: a set of operations available on a component. But it can represent a source of requests, an abstract service provided by the component, or represent a source of or destination for data.

Definition A.15 Component is comprised by ports (interfaces) and their implementations:

$Component = \{Port, C, M, Init\}$

Where *Port* is the set of the interfaces, *C* is the set of classes, *M* is the mapping from *C* to *Port*, describing the input/output parameters of *Port* and *Init* is the initial state of the component.

Each component has a set of interfaces, $Port = \{ Port_1, Port_2, ..., Port_n\}$, each $Port_i$ is independent from other interfaces and is defined as 8-tuple:

$Port_i=\{ID, Prov_i, Requ_i, Attr_i, Beha_i, Msgs_i, Cons_i, Nonf_i\}$

where *ID* is identification of interface, $Prov_i$ is the functional set provided by the *i*th interface, $Requ_i$ is the functional set required from the *i*th interface, $Attr_i$ is the attribute set of the *i*th interface, $Beha_i$ is the behaviour set provided by the *i*th interface, $Msgs_i$ is the message set created by the *i*th interface, $Cons_i$ is the constraint set of the *i*th interface, including the initial states, pre-condition and post-condition, $Nonf_i$ is non-functional specification of the *i*th interface.

Connectors represent communication glue that captures the nature of an interaction between components. Connectors can be used to model a variety of different sorts of interactions under a number of different models. A typical connector might define a synchronisation model for communication, a communication protocol, a locking model,

or characteristics of a communication channel like bandwidth. An important feature of architectural modelling is the fact that interactions are an explicit first class concept. This is in contrast to many object-oriented design approaches where communication is often implicit in the description of classes/objects/components. Intuitively, connectors correspond to the "lines" in box and line diagrams [1].

Definition A.16 Connector implements the connection of components by modelling the interaction rules between the components. Connector is 6-tuple:

Connector={ID,Role,Beha,Msgs,Cons,Nfun}.

where *ID* is identification of connector, *Role* is the set of interaction points between component and connector, *Beha* is the behaviour set provided by connector, *Msgs* is the message set created by the Role of Connector, Cons is the constraint set of the connector, including the initial states, pre-condition and post-condition, *Nfun* is non-functional specification of connector.

A connector includes a set of interfaces in the form of roles, each of which defines a participant in the interaction captured by the connector. A simple role can be seen as an interface to a communication channel, defining an interface to the connector in the same way a port provides an interface to a component. Each role in *Role* set is defined as: *role={Id, Action, Event, Rcons}*, where *Id* is the identification of role, *Action* is the action set of the role, *Event* is the set of events created by *Role*, *Rcons* is the constraint set of *Role*.

A.2 Lexical Conventions

microsyntax ::= lexeme (lexeme)*
lexeme ::= blank | comment | token
blank ::= white-space | end-of-line
white-space ::= space | tab
end-of-line ::= newline | carriage-return | carriage-return newline
comment ::= complete-comment | abbr-comment
complete-comment ::= COMMENT: "comment-body";
abbr-comment ::= C: "comment-body";
comment-body ::= (non-quotes-character)*
non-quotes-character ::= any character except " and "

token ::= identifier | keyword | special-symbol | literal
identifier ::= letter (letter | digit)*
letter ::= _, $, a through z, or A through Z
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
keyword ::= wsl-reserved-word | wml-keyword
wml-keyword ::= cml-reserved-word | adl-reserved-word | fdl-reserved-word
... ...
special-symbol ::= wsl-special-symbol | wml-special-symbol
... ...
literal ::= integer-literal | floating-point-literal | boolean-literal | character-literal | string-literal | null-literal
... ...

A.3 Syntax for WSL in WML

WSL ::= statements <EOF>
statements ::= statement (<S_SEMICOLON> statement)*
statement ::= stat_if
| stat_d_if
| stat_d_do
| stat_while
| stat_do
| stat_exit
| stat_for
| stat_var
| stat_comment
| stat_assert
| stat_assignment
| stat_push
| stat_pop
| stat_join
| stat_actions
| stat_call
| stat_print
| stat_mw_func_decl
| stat_begin
| stat_foreach
| stat_ateach
| stat_ifmatch
| stat_ifmatch2
| stat_maphash
| stat_error
| stat_spec

| stat_single_assign
| stat_pattern
| stat_proc_call
| (<S_SKIP>)
| (<S_ABORT>)
| (<S_STAT_PLACE>)

stat_if ::= ((<S_IF> condition <S_THEN> statements) ((<S_ELSIF> condition <S_THEN> statements))* ((<S_ELSE> statements <S_FI>) | pseudo_else <S_FI>))

stat_d_if ::= ((<S_D_IF> condition <S_ARROW> statements) ((<S_BOX> condition <S_ARROW> statements))* <S_FI>)

stat_d_do ::= ((<S_D_DO> condition <S_ARROW> statements) ((<S_BOX> condition <S_ARROW> statements))* <S_OD>)

stat_while ::= (<S_WHILE> condition <S_DO> statements <S_OD>)

stat_do ::= (<S_DO> statements <S_OD>)

stat_exit ::= T_Exit

stat_for ::= (<S_FOR> T_Var_Lvalue <S_BECOMES> expression <S_TO> expression <S_STEP> expression <S_DO> statements <S_OD>)
| (<S_FOR> T_Var_Lvalue <S_IN> s_expression <S_DO> statements <S_OD>)

stat_var ::= (<S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON> statements <S_ENDVAR>)

stat_comment ::= T_Comment

stat_assert ::= (<S_LBRACE> condition <S_RBRACE>)

Stat_assignment ::= (<S_LANGLE> assigns <S_RANGLE>)

stat_push ::= (<S_PUSH> <S_LPAREN> T_Var_Lvalue <S_COMMA> s_expression <S_RPAREN>)

stat_pop ::= (<S_POP> <S_LPAREN> T_Var_Lvalue <S_COMMA> T_Var_Lvalue <S_RPAREN>)

stat_join ::= (<S_JOIN> statements <S_COMMA> statements <S_ENDJOIN>)

stat_actions ::= (<S_ACTIONS> T_IdentifierName <S_COLON> actions <S_ENDACTIONS>)

stat_call ::= T_Call

stat_print ::= (<S_PRINT> <S_LPAREN> expressions <S_RPAREN>)
| (<S_PRINFLUSH> <S_LPAREN> expressions <S_RPAREN>)

stat_mw_func_decl ::= (<S_MW_PROC> T_AtName <S_LPAREN> ((lvalue (<S_COMMA> lvalue)*)*) var_lvalues <S_RPAREN> <S_DEFINE> statements (<S_END> | <S_FULLSTOP>))
| (<S_MW_FUNCT> T_AtName <S_LPAREN> ((lvalue (<S_COMMA> lvalue)*)*) <S_RPAREN> <S_DEFINE> ((<S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON> statements <S_SEMICOLON> <S_LPAREN> expression <S_RPAREN> (<S_END> | <S_FULLSTOP>)) | (<S_COLON> statements <S_SEMICOLON> <S_LPAREN> expression <S_RPAREN> (<S_END> | <S_FULLSTOP>))))
| (<S_MW_BFUNCT> T_AtName <S_QUERY> <S_LPAREN>

((lvalue (<S_COMMA> lvalue)*)*) <S_RPAREN> <S_DEFINE> ((<S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON> statements <S_SEMICOLON> <S_LPAREN> condition <S_RPAREN> (<S_END> | <S_FULLSTOP>)) | (<S_COLON> statements <S_SEMICOLON> <S_LPAREN> condition <S_RPAREN> (<S_END> | <S_FULLSTOP>))))

stat_begin ::= (<S_BEGIN> statements <S_WHERE> defines <S_END>)

stat_foreach ::= <S_FOREACH> ((<S_STATEMENT> <S_DO> statements <S_OD>) | (<S_STATEMENTS> <S_DO> statements <S_OD>) | (<S_VARIABLE> <S_DO> statements <S_OD>) | (<S_GLOBAL> <S_VARIABLE> <S_DO> statements <S_OD>) | (<S_LVALUE> <S_DO> statements <S_OD>) | (<S_STS> <S_DO> statements <S_OD>) | (<S_NAS> <S_DO> statements <S_OD>) | (<S_EXPRESSION> <S_DO> statements <S_OD>) | (<S_CONDITION> <S_DO> statements <S_OD>) | (<S_TERMINAL> ((<S_STATEMENT> <S_DO> statements <S_OD>) | (<S_STATEMENTS> <S_DO> statements <S_OD>))))

stat_ateach ::= <S_ATEACH> ((<S_STATEMENT> <S_DO> statements <S_OD>) | (<S_STATEMENTS> <S_DO> statements <S_OD>) | (<S_VARIABLE> <S_DO> statements <S_OD>) | (<S_GLOBAL> <S_VARIABLE> <S_DO> statements <S_OD>) | (<S_LVALUE> <S_DO> statements <S_OD>) | (<S_STS> <S_DO> statements <S_OD>) | (<S_NAS> <S_DO> statements <S_OD>) | (<S_EXPRESSION> <S_DO> statements <S_OD>) | (<S_CONDITION> <S_DO> statements <S_OD>) | (<S_TERMINAL> ((<S_STATEMENT> <S_DO> statements <S_OD>) | (<S_STATEMENTS> <S_DO> statements <S_OD>))))

stat_ifmatch ::= <S_IFMATCH> ((<S_STATEMENTS> statements <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_STATEMENT> statement <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_EXPRESSION> expression <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_EXPRESSIONS> expressions <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_CONDITION> condition <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_DEFINITION> define <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_DEFINITIONS> defines <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_ASSIGN> assign <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_ASSIGNS> assigns_node <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_ACTION> action <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_GUARDED> guarded <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_LVALUE> lvalue <S_THEN>

statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_LVALUES> lvalues <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))))

stat_ifmatch2 ::= <S_IFMATCH2> ((<S_STATEMENTS> statements <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_STATEMENT> statement <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_EXPRESSION> expression <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_EXPRESSIONS> expressions <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_CONDITION> condition <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_DEFINITION> define <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_DEFINITIONS> defines <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_ASSIGN> assign <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_ASSIGNS> assigns_node <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_ACTION> action <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_GUARDED> guarded <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_LVALUE> lvalue <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))) | (<S_LVALUES> lvalues <S_THEN> statements (((<S_ENDMATCH>)) | (<S_ELSE> statements <S_ENDMATCH>))))

stat_maphash ::= (<S_MAPHASH> <S_LPAREN> T_Name <S_COMMA> expression <S_RPAREN>)

stat_error ::= (<S_ERROR> <S_LPAREN> expressions <S_RPAREN>)

stat_spec ::= (<S_SPEC> <S_LANGLE> lvalues <S_RANGLE> <S_COLON> condition <S_ENDSPEC>)

stat_proc_call ::= (T_IdentifierName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) var_lvalues <S_RPAREN>)

| (<S_PLINK_P> T_IdentifierName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) var_lvalues <S_RPAREN>)

| (T_AtName (<S_LPAREN>)* ((expression (<S_COMMA> expression)*)*) var_lvalues (<S_RPAREN>)*)

| (T_AtPatOneName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) var_lvalues <S_RPAREN>)

| (<S_PLINK_XP> T_IdentifierName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)

stat_pattern ::= T_Stat_Pat_One

| T_Stat_Pat_Many

| T_Stat_Pat_Any

stat_single_assign ::= ((lvalue <S_BECOMES> expression))

```
                | ( lvalue <S_FULLSTOP> <S_LPAREN> expression <S_RPAREN>
                  <S_BECOMES> expression )
      guarded ::= ( condition ( <S_THEN> | <S_ARROW> ) statements )
                | ( ( T_Cond_Pat_One | T_Cond_Pat_Many | T_Cond_Pat_Any )
                  ( <S_THEN> | <S_ARROW> ) statements )
      defines ::= define ( ( <S_COMMA> define ) | ( define ) )*
       define ::= ( stat_func_decl | T_Defn_Pat_One | T_Defn_Pat_Many |
                  T_Defn_Pat_Any )
stat_func_decl ::= ( <S_PROC> T_IdentifierName <S_LPAREN> ( ( lvalue
                  ( <S_COMMA> lvalue )* )* ) var_lvalues <S_RPAREN> <S_DEFINE>
                  statements ( <S_END> | <S_FULLSTOP> ) )
                | ( <S_FUNCT> T_IdentifierName <S_LPAREN> ( ( lvalue
                  ( <S_COMMA> lvalue )* )* ) <S_RPAREN> <S_DEFINE>
                  ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON>
                  <S_LPAREN> expression <S_RPAREN> ( <S_END> |
                  <S_FULLSTOP> ) ) | ( <S_COLON> <S_LPAREN> expression
                  <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
                | ( <S_BFUNCT> T_IdentifierName <S_QUERY> <S_LPAREN>
                  ( ( lvalue ( <S_COMMA> lvalue )* )* ) <S_RPAREN> <S_DEFINE>
                  ( ( <S_VAR> <S_LANGLE> assigns <S_RANGLE> <S_COLON>
                  <S_LPAREN> condition <S_RPAREN> ( <S_END> |
                  <S_FULLSTOP> ) ) | ( <S_COLON> <S_LPAREN> condition
                  <S_RPAREN> ( <S_END> | <S_FULLSTOP> ) ) ) )
      actions ::= ( action ( action )* )
       action ::= ( ( T_IdentifierName | T_Action_Pat_One | T_Action_Pat_Many |
                  T_Action_Pat_Any ) <S_DEFINE> statements ( <S_END> |
                  <S_DOTSPACE> ) )
assigns_node ::= ( assign ( <S_COMMA> assign )* )
      assigns ::= assign ( <S_COMMA> assign )*
       assign ::= ( T_Var_Lvalue <S_BECOMES> expression )
   var_lvalues ::= ( ( <S_VAR> ( lvalue ( <S_COMMA> lvalue )* )* )* )
      lvalues ::= ( lvalue ( <S_COMMA> lvalue )* )
       lvalue ::= ( T_Var_Lvalue | T_Lvalue_Pat_One | T_Lvalue_Pat_Many |
                  T_Lvalue_Pat_Any ) ( <S_LBRACKET> a_expressions
                  <S_RBRACKET> | <S_LBRACKET> a_expression <S_DOTDOT>
                  ( <S_RBRACKET> | a_expression <S_RBRACKET> ) |
                  <S_LBRACKET> a_expression <S_COMMA> a_expression
                  <S_RBRACKET> | ( T_Struct_Lvalue ) )*
    condition ::= ( b_term ( <S_OR> b_term )* )
       b_term ::= ( b_factor ( <S_AND> b_factor )* )
     b_factor ::= ( <S_NOT> b_factor )
                | b_atom
       b_atom ::= ( <S_LPAREN> condition <S_RPAREN> )
                | ( <S_TRUE> )
                | ( <S_FALSE> )
                | ( <S_COND_PLACE> )
                | rel_exp
```

| cond_pat
| cond_prefix

cond_pat ::= T_Cond_Pat_One
| T_Cond_Pat_Many
| T_Cond_Pat_Any

rel_exp ::= expression (<S_EQUAL> expression | <S_NEQ> expression | <S_LANGLE> expression | <S_RANGLE> expression | <S_LEQ> expression | <S_GEQ> expression | <S_IN> expression | <S_NOTIN> expression)

cond_prefix ::= (<S_EVEN> <S_QUERY> <S_LPAREN> expression <S_RPAREN>)
| (<S_ODD> <S_QUERY> <S_LPAREN> expression <S_RPAREN>)
| (<S_SUBSET> <S_QUERY> <S_LPAREN> s_expression <S_COMMA> s_expression <S_RPAREN>)
| (<S_MEMBER> <S_QUERY> <S_LPAREN> expression <S_COMMA> s_expression <S_RPAREN>)
| (T_IdentifierName <S_QUERY> <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)
| (T_AtName <S_QUERY> <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)
| (<S_PLINK_XC> T_IdentifierName <S_QUERY> <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)

expressions ::= (expression (<S_COMMA> expression)*)

expression ::= a_expression
| fill_expression
| fill2_expression
| if_expression

if_expression ::= (<S_IF> condition <S_THEN> expression <S_ELSE> expression <S_FI>)

fill_expression ::= (<S_FILL> <S_STATEMENTS> statements <S_ENDFILL>)
| (<S_FILL> <S_STATEMENT> statement <S_ENDFILL>)
| (<S_FILL> <S_EXPRESSION> expression <S_ENDFILL>)
| (<S_FILL> <S_EXPRESSIONS> expressions <S_ENDFILL>)
| (<S_FILL> <S_CONDITION> condition <S_ENDFILL>)
| (<S_FILL> <S_DEFINITION> define <S_ENDFILL>)
| (<S_FILL> <S_DEFINITIONS> defines <S_ENDFILL>)
| (<S_FILL> <S_ASSIGN> assign <S_ENDFILL>)
| (<S_FILL> <S_ASSIGNS> assigns_node <S_ENDFILL>)
| (<S_FILL> <S_ACTION> action <S_ENDFILL>)
| (<S_FILL> <S_GUARDED> guarded <S_ENDFILL>)
| (<S_FILL> <S_LVALUE> lvalue <S_ENDFILL>)
| (<S_FILL> <S_LVALUES> lvalues <S_ENDFILL>)

fill2_expression ::= (<S_FILL2> <S_STATEMENTS> statements <S_ENDFILL>)
| (<S_FILL2> <S_STATEMENT> statement <S_ENDFILL>)
| (<S_FILL2> <S_EXPRESSION> expression <S_ENDFILL>)

| (<S_FILL2> <S_EXPRESSIONS> expressions <S_ENDFILL>)
| (<S_FILL2> <S_CONDITION> condition <S_ENDFILL>)
| (<S_FILL2> <S_DEFINITION> define <S_ENDFILL>)
| (<S_FILL2> <S_DEFINITIONS> defines <S_ENDFILL>)
| (<S_FILL2> <S_ASSIGN> assign <S_ENDFILL>)
| (<S_FILL2> <S_ASSIGNS> assigns_node <S_ENDFILL>)
| (<S_FILL2> <S_ACTION> action <S_ENDFILL>)
| (<S_FILL2> <S_GUARDED> guarded <S_ENDFILL>)
| (<S_FILL2> <S_LVALUE> lvalue <S_ENDFILL>)
| (<S_FILL2> <S_LVALUES> lvalues <S_ENDFILL>)

S_expression ::= a_expression

A_expressions ::= a_expression

A_expression ::= term (<S_PLUS> term | <S_MINUS> term | <S_CONCAT> term | <S_UNION> term)*

term ::= factor (<S_TIMES> factor | <S_SLASH> factor | <S_MOD> factor | <S_DIV> factor | <S_BACKSLASH> factor)*

factor ::= (true_factor | (<S_MINUS> factor)) (<S_CARET> a_expression | <S_CARET> <S_CARET> s_expression | <S_LBRACKET> a_expressions <S_RBRACKET> | <S_LBRACKET> a_expression <S_DOTDOT> (<S_RBRACKET> | a_expression <S_RBRACKET>) | <S_LBRACKET> a_expression <S_COMMA> a_expression <S_RBRACKET> | (T_Struct) | <S_FULLSTOP> <S_LPAREN> expression <S_RPAREN>)*

true_factor ::= exp_atom (<S_EXPONENT> factor)*

exp_atom ::= <S_LPAREN> a_expression <S_RPAREN>
| T_Number
| a_prefix_op
| (T_IdentifierName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)
| (T_AtName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)
| (<S_PLINK_XF> T_IdentifierName <S_LPAREN> ((expression (<S_COMMA> expression)*)*) <S_RPAREN>)
| <S_EXPN_PLACE>
| <S_VAR_PLACE>
| T_Expn_Pat_One
| T_Expn_Pat_Many
| T_Expn_Pat_Any
| T_Variable
| T_String
| T_Set
| T_Sequence
| numb_type
| s_prefix_op

a_prefix_op ::= (<S_ABS> <S_LPAREN> a_expression <S_RPAREN>)

| (<S_FRAC> <S_LPAREN> a_expression <S_RPAREN>)
| (<S_INT> <S_LPAREN> a_expression <S_RPAREN>)
| (<S_SGN> <S_LPAREN> a_expression <S_RPAREN>)
| (<S_MAX> <S_LPAREN> a_expression <S_COMMA> a_expression <S_RPAREN>)
| (<S_MIN> <S_LPAREN> a_expression <S_COMMA> a_expression <S_RPAREN>)
| (<S_LENGTH> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_REDUCE> <S_LPAREN> T_Name <S_COMMA> s_expression <S_RPAREN>)
| (<S_HEAD> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_LAST> <S_LPAREN> s_expression <S_RPAREN>)

T_Set ::= (<S_LBRACE> expression <S_VBAR> condition <S_RBRACE>)

T_Sequence ::= (<S_LANGLE> ((expression (<S_COMMA> expression)*)*) <S_RANGLE>)

numb_type ::= <S_RATS>
| <S_REALS>
| <S_NATS>
| <S_INTS>

s_prefix_op ::= (<S_MAP> <S_LPAREN> T_Name <S_COMMA> s_expression <S_RPAREN>)
| (<S_POWERSET> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_TAIL> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_BUTLAST> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_SLENGTH> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_SUBSTR> <S_LPAREN> expressions <S_RPAREN>)
| (<S_INDEX> <S_LPAREN> expressions <S_RPAREN>)
| (<S_REDUCE> <S_LPAREN> T_Name <S_COMMA> s_expression <S_RPAREN>)
| (<S_HEAD> <S_LPAREN> s_expression <S_RPAREN>)
| (<S_LAST> <S_LPAREN> s_expression <S_RPAREN>)

T_Cond_Pat_One ::= <S_PAT_ONE> <S_IDENTIFIER>

T_Cond_Pat_Many ::= <S_PAT_MANY> <S_IDENTIFIER>

T_Cond_Pat_Any ::= <S_PAT_ANY> <S_IDENTIFIER>

T_Expn_Pat_One ::= <S_PAT_ONE> <S_IDENTIFIER>

T_Expn_Pat_Many ::= <S_PAT_MANY> <S_IDENTIFIER>

T_Expn_Pat_Any ::= <S_PAT_ANY> <S_IDENTIFIER>

T_Lvalue_Pat_One ::= <S_PAT_ONE> <S_IDENTIFIER>

T_Lvalue_Pat_Many ::= <S_PAT_MANY> <S_IDENTIFIER>

T_Lvalue_Pat_Any ::= <S_PAT_ANY> <S_IDENTIFIER>

T_Stat_Pat_One ::= <S_PAT_ONE> <S_IDENTIFIER>

T_Stat_Pat_Many ::= <S_PAT_MANY> <S_IDENTIFIER>

T_Stat_Pat_Any ::= <S_PAT_ANY> <S_IDENTIFIER>

T_Action_Pat_One ::= <S_PAT_ONE> <S_IDENTIFIER>

T_Action_Pat_Many ::= <S_PAT_MANY> <S_IDENTIFIER>
T_Action_Pat_Any ::= <S_PAT_ANY> <S_IDENTIFIER>
T_Defn_Pat_One ::= <S_PAT_ONE> <S_IDENTIFIER>
T_Defn_Pat_Many ::= <S_PAT_MANY> <S_IDENTIFIER>
T_Defn_Pat_Any ::= <S_PAT_ANY> <S_IDENTIFIER>
T_String ::= <S_STRING>
T_Number ::= <S_NUMBER>
T_Variable ::= <S_IDENTIFIER>
T_Name ::= <S_STRING>
T_IdentifierName ::= <S_IDENTIFIER>
T_AtName ::= <S_AT> (<S_IDENTIFIER> | <S_AMBIGOUS_IDENTIFIER>)
T_AtPatOneName ::= <S_AT_PAT_ONE> (<S_IDENTIFIER> | <S_AMBIGOUS_IDENTIFIER>)
T_Var_Lvalue ::= <S_IDENTIFIER>
T_Exit ::= <S_EXIT> <S_LPAREN> <S_NUMBER> <S_RPAREN>
T_Comment ::= <S_COMMENT> <S_STRING>
T_Call ::= <S_CALL> <S_IDENTIFIER>
T_Struct_Lvalue ::= <S_FULLSTOP> T_IdentifierName
T_Struct ::= <S_FULLSTOP> T_IdentifierName

A.4 Syntax for CML in WML

CML ::= <MODEL> Model_name Extension_use (<DIAGRAMS> View_element_decl_list)? (Package_declaration | Subsystem_declaration | Actor_declaration)* <END> <EOF>

Model_name ::= <IDENTIFIER>
View_element_decl_list ::= View_element_declaration ("," View_element_declaration)*
View_element_declaration ::= View_element_name ":" View_element_kind
View_element_name ::= <IDENTIFIER>
View_element_kind ::= <IDENTIFIER>
Package_declaration ::= <PACKAGE> Package_name Extension_use (Viewed_with)? (Package_inheritance)? (Package_import)? (Package_element_decl_list)? <END>
Package_name ::= <IDENTIFIER>
Viewed_with ::= <VIEWED> <WITH> View_element_name_list
View_element_name_list ::= View_element_name (Position)? ("," View_element_name (Position)?)*
Position ::= "(" <INTEGER_LITERAL> "," <INTEGER_LITERAL> ("," <INTEGER_LITERAL>)? ")"
Package_inheritance ::= <INHERIT> Package_name ("," Package_name)*
Package_import ::= <IMPORT> Package_import_elem ("," Package_import_elem)*
Package_import_elem ::= Visibility Element_path (<AS> Alias)? <FROM> Package_name
Element_path ::= Element_name ("::" Element_name)*

Element_name ::= <IDENTIFIER>

Alias ::= Element_name

Package_element_decl_list ::= <IS> (Package_element_decl)+

Package_element_decl ::= (Actor_declaration | CML | Package_declaration | Interface_declaration | Class_declaration | Process_declaration | Thread_declaration | Relation_declaration | Stereotype_declaration | Constraint_declaration | Tagged_values | Activity_model | Collaboration_declaration | Comment_definition | Actor_or_exception Light_body | Component_or_node Ultra_ligth_body)

Comment_definition ::= <TEXT_MULTILINE> <ATTACHED> <TO> Element_name

Actor_or_exception ::= (Actor | Exception_dec)

Actor ::= <ACTOR> Light_body

Exception_dec ::= <EXCEPTION> Light_body

Light_body ::= Element_name (Formal_generics)? (Viewed_with)? Extension_use (Inheritance)? Features <END>

Formal_generics ::= Formal_generic ("," Formal_generic)* "]"

Formal_generic ::= Element_name (Constraint_declaration)?

Inheritance ::= <INHERIT> Parent (";" Parent)*

Parent ::= Class_type (Feature_adaptation)?

Class_type ::= Class_name Actual_generics

Actual_generics ::= Type ("," Type)* "]"

Type ::= (Class_type | Class_type_expanded | Anchored | Bit_type)

Class_type_expanded ::= <EXPANDED> Class_type

Anchored ::= <LIKE> Anchor

Anchor ::= (<IDENTIFIER> | <CURRENT>)

Bit_type ::= <BIT> Constant

Constant ::= (Manifest_constant | Entity)

Manifest_constant ::= (<TRUE> | <FALSE> | <CHARACTER_LITERAL> | <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | <STRING_LITERAL>)

Entity ::= (<IDENTIFIER> | <RESULT> | <CURRENT>)

Feature_adaptation ::= <ADAPTATION> (Rename)? (New_exports)? (Undefine)? (Redefine)? (Select)? <END>

Rename ::= <RENAME> Rename_list

Rename_list ::= Rename_pair "," (Rename_pair)*

Rename_pair ::= Feature_name <AS> Feature_name

Feature_name ::= (Element_name | Prefix | Infix)

Prefix ::= <PREFIX> "(" Prefix_operator ")"

Prefix_operator ::= Unary <IDENTIFIER>

Unary ::= (<NOT> | "+" | "-")

Infix ::= <INFIX> "(" Infix_operator ")"

Infix_operator ::= (Binary | <IDENTIFIER>)

Binary ::= ("+" | "-" | "*" | "/" | "<" | ">" | "<=" | ">=" | "//" | "\\" | <AND> | <OR> | <XOR> | <AND> <THEN> | <OR> <ELSE> | <IMPLIES>)

New_exports ::= <EXPORT> New_export_item (";" New_export_item)*

New_export_item ::= Clients Feature_set

Clients ::= "{" Class_name ("," Class_name)* "}"

Feature_set ::= (Identifier_list | <ALL>)

Undefine ::= <UNDEFINE> (Feature_list)+

Redefine ::= <REDEFINE> (Feature_list)+

Select ::= <SELECT> (Feature_list)+

Component_or_node ::= (Component | Node)

Component ::= <COMPONENT> Ultra_ligth_body

Node ::= <NODE> Ultra_ligth_body

Ultra_ligth_body ::= Element_name (Viewed_with)? Extension_use (Element_name ("," Element_name)*)? <END>

Subsystem_declaration ::= Subsystem_header (Formal_generics)? Extension_use (Viewed_with)? (Inheritance)? (Package_import)? (Interface_operation_declaration)? (Package_element_decl)* <END>

Subsystem_header ::= <SUBSYSTEM> Subsystem_name

Subsystem_name ::= <IDENTIFIER>

Features ::= (<FEATURE> "{" Visibility "}" Feature_list <END>)*

Visibility ::= (<ANY> | <NONE> | Classifier_list)

Classifier_list ::= Classifier_name ("," Classifier_name)*

Classifier_name ::= <IDENTIFIER>

Feature_list ::= Feature_declaration (";" Feature_declaration)*

Feature_declaration ::= (Attribute | Operation_declaration | Method_declaration)

Attribute ::= <IDENTIFIER> Type_mark Extension_use (<IS> Initial_value)?

Initial_value ::= Expression

Expression ::= (<CALL> | Operator_expression | Equality | Manifest_constant | Manifest_array | Old | Strip)

Operator_expression ::= (Parenthesized | Unary_expression | Binary_expression)

Parenthesized ::= "(" Expression ")"

Unary_expression ::= Unary Expression

Binary_expression ::= Element_expression (Binary Expression)*

Element_expression ::= (<CALL> | Unary_expression | Parenthesized | Equality | Manifest_constant | Manifest_array | Old | Strip)

Manifest_array ::= "<<" Expression ("," Expression)* ">>"

Equality ::= Element_equal_expression (Comparision Expression)*

Comparision ::= ("=" | "<>")

Element_equal_expression ::= (<CALL> | Unary_expression | Parenthesized | Manifest_constant | Manifest_array | Old | Strip)

Old ::= <OLD> Expression

Strip ::= <STRIP> "(" Identifier_list ")"

Type_mark ::= ":" (<LIKE>)? <IDENTIFIER> (Formal_generics)?

Operation_declaration ::= New_operation Operation_body

New_operation ::= <OPERATION> <IDENTIFIER>

Operation_body ::= (Formal_arguments)? (Type_mark)? Extension_use (<IS> Specification)?

Formal_arguments ::= "(" Entity_declaration_list ")"

Entity_declaration_list ::= Entity_declaration_group (";" Entity_declaration_group)*

Entity_declaration_group ::= (Parameter_kind)? <IDENTIFIER> ("," <IDENTIFIER>)* Type_mark

Parameter_kind ::= (<IN> | <OUT> | <INOUT>)

Specification ::= <TEXT_MULTILINE>

Method_declaration ::= New_method Method_body

New_method ::= <IDENTIFIER>

Method_body ::= (Formal_arguments)? (Type_mark)? Extension_use (Specification_use)? (<IS> Routine)?

Specification_use ::= (<IDENTIFIER> | <TEXT_MULTILINE>)

Routine ::= <TEXT_MULTILINE>

Actor_declaration ::= Actor_header (Formal_generics)? (Viewed_with)? Extension_use (Inheritance)? (Interface_operation_declaration)* <END>

Actor_header ::= <ACTOR> <IDENTIFIER>

Interface_declaration ::= Interface_header (Formal_generics)? (Viewed_with)? Extension_use (Inheritance)? (Interface_operation_declaration)* <END>

Interface_header ::= <INTERFACE> <IDENTIFIER>

Interface_operation_declaration ::= <FEATURE> "{" Visibility "}" (Operation_declaration)*

Process_declaration ::= <ProcessClass> Process_name ("(" (Init_Parameter)? ")")* ("Extends" Process_name)? (<Port> (Port_name)?)* (<message> (Message_Sig)?)* (<method> (Process_Method)?)* <END>

Process_name ::= <IDENTIFIER>

Init_Parameter ::= Declarations

Declarations ::= Declaration ("," Declaration)*

Declaration ::= <IDENTIFIER> ":" <IDENTIFIER>

Port_name ::= <IDENTIFIER>

Message_Sig ::= Message_name ":" Port_name ":" Process_name

Message_name ::= <IDENTIFIER>

Process_Method ::= Method_name "(" Declarations ")"

Thread_declaration ::= <Thread> Thread_name "In" Process_name ("(" (Init_Parameter)? ")")*

Class_declaration ::= Class_header (Formal_generics)? (Viewed_with)? Extension_use (Inheritance)? (State_machine)? Features (Use_of_constraint)? <END>

Class_header ::= (<DEFERRED>)? <CLASS> Class_name

Class_name ::= <IDENTIFIER>

Extension_use ::= (Comment)? (Use_of_stereotype)? (Use_of_constraint | Use_of_tagged_value)*

Comment ::= <TEXT_MULTILINE>

Constraint_declaration ::= <CONSTRAINT> (Constraint_def)+ <END>

Constraint_def ::= <IDENTIFIER> (Use_of_stereotype | Use_of_tagged_value)* <IS> Constraint_expression

Constraint_expression ::= <TEXT_MULTILINE>

Use_of_constraint ::= <CONSTRAINED> <BY> Constraint_body (Logical_connector Constraint_body)*

Constraint_body ::= (<IDENTIFIER> | Constraint_expression)

Logical_connector ::= (<AND> | <OR> | <XOR>)

Tagged_values ::= <TAG> <VALUES> (Tagged_values_def)+ <END>

Tagged_values_def ::= <IDENTIFIER> (<IS> <STRING_LITERAL>)?

Use_of_tagged_value ::= <WITH> <TAG> <VALUES> "(" Property ("," Property)* ")"

Property ::= "<" <IDENTIFIER> ("," <STRING_LITERAL>)? ">"

Stereotype_declaration ::= <STEREOTYPE> <IDENTIFIER> <OF> Base_class (Icon)? (Stereotype_parent)? (Constraint_declaration | Tagged_values)* <END>

Base_class ::= <IDENTIFIER>

Icon ::= <VIEWED> <AS> <STRING_LITERAL>

Stereotype_parent ::= <INHERIT> Identifier_list

Use_of_stereotype ::= <STEREOTYPED> <WITH> <IDENTIFIER>

Collaboration_declaration ::= <COLLABORATION> Collaboration_name (Formal_generics)? (Viewed_with)? Extension_use (Implementation_of)? (Class_declaration | Interface_declaration | Relation_declaration)* (Action_definition)* (Message ("," Message)*)? <END>

Collaboration_name ::= Element_name

Implementation_of ::= <IMPLEMENTS> Classifier_or_operation

Classifier_or_operation ::= Element_name

Message ::= <ACTIONS> (Action_definition)+ <TO> Classifier_name <FROM> Classifier_name

Relation_declaration ::= (<DEFERRED>)? <RELATION> Relation_name (Formal_generics)? (Relation_inheritance)? (Link_list)? (Delegation_list)? Features <END>

Relation_name ::= Element_name

Relation_inheritance ::= <INHERIT> Parent_relation (";" Parent_relation)*

Parent_relation ::= Relation_type (<ADAPTATION> Relation_feature_adaptation)?

Relation_type ::= Relation_path (Actual_generics)?

Relation_path ::= Element_path

Relation_feature_adaptation ::= (Rename)? (New_exports)? (Undefine)? (Relation_redefine)? (Select)? <END>

Relation_redefine ::= <REDEFINE> (Feature_list | Redefine_with_list)

Redefine_with_list ::= Redefine_pair ("," Redefine_pair)*

Redefine_pair ::= Feature_name <WITH> Feature_name

Link_list ::= <LINK> (Type_link_two_list | Dependency ("," Dependency)*)

Type_link_two_list ::= Type_link ("," Type_link)+

Type_link ::= (Classifier_name | <THIS>)

Dependency ::= Element_path <TO> Element_path

Delegation_list ::= Delegation (Deleg_relation_redefine)? ("," Delegation (Deleg_relation_redefine)?)*

Deleg_relation_redefine ::= Relation_redefine <END>

Delegation ::= (Type_link)? "." Feature_name <LIKE> (Type_link)? "." Feature_name

State_machine ::= <STATE> <MACHINE> Element_name (Viewed_with)? (Constraint_declaration)? (Use_of_constraint)? (Machine_body)? <END>

Machine_body ::= Composite_state (Transition_definition)* (Action_definition)*

State_definition ::= <STATE> Element_name (Viewed_with)? (Constraint_declaration)? Action (Internal_transition)* (Deferred_events)?

Action ::= (<ENTRY> Identifier_list)? (<EXIT> Identifier_list)?

Composite_state ::= <COMPOSITE> State_definition (Concurrent_state_list | State_list)

Concurrent_state_list ::= (<CONCURRENT> State_list)+

State_list ::= (State_kind <END>)+

```
State_kind ::= ( Simple_state | Pseudostate | Submachine | Composite_state )
Simple_state ::= <SIMPLE> State_definition
Pseudostate ::= Pseudostate_kind <STATE> Element_name
                ( Constraint_declaration )? Action
```

A.5 Syntax of ADL in WML

```
WML_ADL ::= ( <IMPORT> ( Filename ";" | <STRING_LITERAL> ";" ) )*
            ( TypeDeclaration | FamilyDeclaration |
            DesignAnalysisDeclaration | PropertyDeclaration |
            PropertiesBlock | SystemDeclaration )* <EOF>
Filename ::= ( "$" | "%" )? <IDENTIFIER> ( ( ( "." | ":" | "-" | "+" | "\\" |
             "\\\\" | "/" | "$" | "%" ) )+ <IDENTIFIER> )*
FamilyDeclaration ::= ( <FAMILY> | <STYLE> ) <IDENTIFIER> ( ";" | ( "="
                      FamilyBody ( ";" )? ) | ( <EXTENDS>
                      lookup_SystemTypeByName ( "," lookup_SystemTypeByName )*
                      <WITH> FamilyBody ( ";" )? ) )
FamilyBody ::= "{" "}"
             | "{" ( TypeDeclaration | SystemStructure )+ "}"
TypeDeclaration ::= ElementTypeDeclaration
                  | PropertyTypeDeclaration
ElementTypeDeclaration ::= ElementProtoTypeDeclaration
                         | ComTypeDeclaration
                         | GroupTypeDeclaration
                         | ConnectorTypeDeclaration
                         | PortTypeDeclaration
                         | RoleTypeDeclaration
ElementProtoTypeDeclaration ::= ( <ELEMENT> <TYPE> <IDENTIFIER> ( "="
                                parse_EleProtoTypeDestion ( ";" )? | ";" ) | <ELEMENT>
                                <TYPE> <IDENTIFIER> <EXTENDS>
                                lookup_ComTypeByName ( "," lookup_ComTypeByName )*
                                <WITH> parse_EleProtoTypeDes ( ";" )? )
ComTypeDeclaration ::= ( <COMPONENT> <TYPE> <IDENTIFIER> ( "="
                       parse_ComDescription ( ";" )? | ";" ) | <COMPONENT>
                       <TYPE> <IDENTIFIER> <EXTENDS>
                       lookup_ComTypeByName ( "," lookup_ComTypeByName )*
                       <WITH> parse_ComDescription ( ";" )? )
GroupTypeDeclaration ::= ( <GROUP> <TYPE> <IDENTIFIER> ( "="
                         parse_GroupDescription ( ";" )? | ";" ) | <GROUP> <TYPE>
                         <IDENTIFIER> <EXTENDS> lookup_GroupTypeByName
                         ( "," lookup_GroupTypeByName )* <WITH>
                         parse_GroupDescription ( ";" )? )
ConnectorTypeDeclaration ::= ( <CONNECTOR> <TYPE> <IDENTIFIER> ( "="
                           parse_ConnectorDescription ( ";" )? | ";" ) | <CONNECTOR>
                           <TYPE> <IDENTIFIER> <EXTENDS>
                           lookup_ConnectorTypeByName ( ","
                           lookup_ConnectorTypeByName )* <WITH>
```

```
                                parse_ConnectorDescription ( ";" )? )
        PortTypeDeclaration ::= ( <PORT> <TYPE> <IDENTIFIER> ( "="
                                parse_PortDescription ( ";" )? | ";" ) | <PORT> <TYPE>
                                <IDENTIFIER> <EXTENDS> lookup_PortTypeByName ( ","
                                lookup_PortTypeByName )* <WITH> parse_PortDescription
                                ( ";" )? )
        RoleTypeDeclaration ::= ( <ROLE> <TYPE> <IDENTIFIER> ( "="
                                parse_RoleDescription ( ";" )? | ";" ) | <ROLE> <TYPE>
                                <IDENTIFIER> <EXTENDS> lookup_RoleTypeByName ( ","
                                lookup_RoleTypeByName )* <WITH> parse_RoleDescription
                                ( ";" )? )
  lookup_SystemTypeByName ::= <IDENTIFIER>
     lookup_ComTypeByName ::= ( <IDENTIFIER> "." )? <IDENTIFIER>
   lookup_GroupTypeByName ::= ( <IDENTIFIER> "." )? <IDENTIFIER>
lookup_ConnectorTypeByName ::= ( <IDENTIFIER> "." )? <IDENTIFIER>
    lookup_PortTypeByName ::= ( <IDENTIFIER> "." )? <IDENTIFIER>
    lookup_RoleTypeByName ::= ( <IDENTIFIER> "." )? <IDENTIFIER>
lookup_PropertyTypeByName ::= ( <IDENTIFIER> "." )? <IDENTIFIER>
lookup_arbitraryTypeByName ::= ( PropertyTypeDescription | <SYSTEM> | <COMPONENT> |
                                <GROUP> | <CONNECTOR> | <PORT> | <ROLE> |
                                <PROPERTY> | <REPRESENTATION> |
                                NonPropertySetTypeExpression )
      SystemDeclaration ::= <SYSTEM> <IDENTIFIER> ( ":" lookup_SystemTypeByName
                                ( "," lookup_SystemTypeByName )* )? ( "=" SystemBody ( ";" )? |
                                ";" )
            SystemBody ::= ( <NEW> lookup_SystemTypeByName ( ","
                                lookup_SystemTypeByName )* | "{" "}" | "{"
                                ( SystemStructure )+ "}" ) ( <EXTENDED> <WITH>
                                SystemBody )?
       SystemStructure ::= ComDeclaration
                         | ComsBlock
                         | GroupDeclaration
                         | ConnectorDeclaration
                         | ConnectorsBlock
                         | PortDeclaration
                         | PortsBlock
                         | RoleDeclaration
                         | RolesBlock
                         | PropertyDeclaration
                         | PropertiesBlock
                         | AttachmentsDeclaration
                         | RepresentationDeclaration
                         | DesignAnalysisDeclaration
                         | parse_DesignRule
parse_EleProtoTypeDes ::= "{" ( PropertyDeclaration | PropertiesBlock |
                                RepresentationDeclaration )* "}"
```

```
GroupDeclaration ::= <GROUP> <IDENTIFIER> ( ":" lookup_GroupTypeByName
                     ( "," lookup_GroupTypeByName )* )? ( "="
                     parse_GroupDescription ";" | ";" )
parse_GroupDescription ::= ( <NEW> lookup_GroupTypeByName ( ","
                     lookup_GroupTypeByName )* | "{" ( MembersBlock |
                     PropertyDeclaration | PropertiesBlock | parse_DesignRule )*
                     "}" ) ( <EXTENDED> <WITH> parse_GroupDescription )?
ComDeclaration ::= <COMPONENT> <IDENTIFIER> ( ":"
                     lookup_ComTypeByName ( "," lookup_ComTypeByName )* )?
                     ( "=" parse_ComDescription ";" | ";" )
ComsBlock ::= <COMPONENTS> "{" ( <IDENTIFIER> ( ":"
                     lookup_ComTypeByName ( "," lookup_ComTypeByName )* )?
                     ( "=" parse_ComDescription ";" | ";" ) )* "}" ( ";" )?
parse_ComDescription ::= ( <NEW> lookup_ComTypeByName ( ","
                     lookup_ComTypeByName )* | "{" ( PortDeclaration | PortsBlock
                     | PropertyDeclaration | PropertiesBlock |
                     RepresentationDeclaration | parse_DesignRule )* "}" )
                     ( <EXTENDED> <WITH> parse_ComDescription )?
ConnectorDeclaration ::= <CONNECTOR> <IDENTIFIER> ( ":"
                     lookup_ConnectorTypeByName ( ","
                     lookup_ConnectorTypeByName )* )? ( "="
                     parse_ConnectorDescription ";" | ";" )
ConnectorsBlock ::= <CONNECTORS> "{" ( <IDENTIFIER> ( ":"
                     lookup_ConnectorTypeByName ( ","
                     lookup_ConnectorTypeByName )* )? ( "="
                     parse_ConnectorDescription ";" | ";" ) )* "}" ( ";" )?
parse_ConnectorDescription ::= ( <NEW> lookup_ConnectorTypeByName ( ","
                     lookup_ConnectorTypeByName )* | "{" ( RoleDeclaration |
                     RolesBlock | PropertyDeclaration | PropertiesBlock |
                     RepresentationDeclaration | parse_DesignRule )* "}" )
                     ( <EXTENDED> <WITH> parse_ConnectorDescription )?
PortDeclaration ::= <PORT> <IDENTIFIER> ( ":" lookup_PortTypeByName ( ","
                     lookup_PortTypeByName )* )? ( "=" parse_PortDescription ";" |
                     ";" )
PortsBlock ::= <PORTS> "{" ( <IDENTIFIER> ( ":"
                     lookup_PortTypeByName ( "," lookup_PortTypeByName )* )?
                     ( "=" parse_PortDescription ";" | ";" ) )* "}" ( ";" )?
parse_PortDescription ::= ( <NEW> lookup_PortTypeByName ( ","
                     lookup_PortTypeByName )* | "{" ( PropertyDeclaration |
                     PropertiesBlock | RepresentationDeclaration |
                     parse_DesignRule )* "}" ) ( <EXTENDED> <WITH>
                     parse_PortDescription )?
RoleDeclaration ::= <ROLE> <IDENTIFIER> ( ":" lookup_RoleTypeByName ( ","
                     lookup_RoleTypeByName )* )? ( "=" parse_RoleDescription ";" |
                     ";" )
MembersBlock ::= <MEMBERS> "{" ( QualifiedReference ( ";" ) )* "}" ( ";" )?
QualifiedReference ::= <IDENTIFIER> ( ( "." <IDENTIFIER> ) )*
RolesBlock ::= <ROLES> "{" ( <IDENTIFIER> ( ":"
                     lookup_RoleTypeByName ( "," lookup_RoleTypeByName )* )?
                     ( "=" parse_RoleDescription ";" | ";" ) )* "}" ( ";" )?
```

```
parse_RoleDescription ::= ( <NEW> lookup_RoleTypeByName ( ","
                          lookup_RoleTypeByName )* | "{" ( PropertyDeclaration |
                          PropertiesBlock | RepresentationDeclaration |
                          parse_DesignRule )* "}" ) ( <EXTENDED> <WITH>
                          parse_RoleDescription )?
AttachmentsDeclaration ::= ( ( <ATTACHMENTS> "{" ( <IDENTIFIER> "."
                          <IDENTIFIER> "to" <IDENTIFIER> "." <IDENTIFIER>
                          ( "{" ( PropertyDeclaration | PropertiesBlock )* "}" )? ";" )* "}"
                          ( ";" )? ) | ( <ATTACHMENT> <IDENTIFIER> "."
                          <IDENTIFIER> "to" <IDENTIFIER> "." <IDENTIFIER>
                          ( "{" ( PropertyDeclaration | PropertiesBlock )* "}" )? ";" ) )
PropertyDeclaration ::= <PROPERTY> parse_PropertyDescription ";"
PropertiesBlock ::= <PROPERTIES> "{" ( parse_PropertyDescription ( ";"
                          parse_PropertyDescription | ";" )* )? "}" ( ";" )?
parse_PropertyDescription ::= ( <PROPERTY> )? <IDENTIFIER> ( ":"
                          PropertyTypeDescription )? ( "=" PropertyValueDeclaration )?
                          ( <PROPBEGIN> parse_PropertyDescription ( ";"
                          parse_PropertyDescription | ";" )* <PROPEND> |
                          <PROPBEGIN> <PROPEND> )?
PropertyTypeDeclaration ::= <PROPERTY> <TYPE> <IDENTIFIER> ( "=" ( <INT> ";" |
                          <FLOAT> ";" | <STRING> ";" | <BOOLEAN> ";" |
                          <ENUM> ( "{" <IDENTIFIER> ( "," <IDENTIFIER> )*
                          "}" )? ";" | <SET> ( "{" "}" )? ";" | <SET> "{"
                          PropertyTypeDescription "}" ";" | <SEQUENCE> ( "<" ">" )?
                          ";" | <SEQUENCE> "<" PropertyTypeDescription ">" ";" |
                          <RECORD> "[" parse_RecordFieldDescription ( ";"
                          parse_RecordFieldDescription | ";" )* "]" ";" | <RECORD> ( "["
                          "]" )? ";" | <IDENTIFIER> ";" ) )
PropertyTypeDescription ::= <ANY>
                          | <INT>
                          | <FLOAT>
                          | <STRING>
                          | <BOOLEAN>
                          | <SET> ( "{" ( PropertyTypeDescription )? "}" )?
                          | <SEQUENCE> ( "<" ( PropertyTypeDescription )? ">" )?
                          | <RECORD> "[" parse_RecordFieldDescription ( ";"
                            parse_RecordFieldDescription | ";" )* "]"
                          | <RECORD> ( "[" "]" )?
                          | <ENUM> ( "{" <IDENTIFIER> ( "," <IDENTIFIER> )*
                            "}" )?
                          | <ENUM> ( "{" "}" )?
                          | lookup_PropertyTypeByName
parse_RecordFieldDescription ::= <IDENTIFIER> ( "," <IDENTIFIER> )* ( ":"
                          PropertyTypeDescription )?
PropertyValueDeclaration ::= <INTEGER_LITERAL>
                          | <FLOATING_POINT_LITERAL>
                          | <STRING_LITERAL>
                          | <FALSE>
```

```
                    | <TRUE>
                    | WML_ADLSetValue
                    | WML_ADLSequenceValue
                    | WML_ADLRecordValue
                    | <IDENTIFIER>
        WML_ADLSetValue ::= "{" "}"
                    | "{" PropertyValueDeclaration ( "," PropertyValueDeclaration )*
                      "}"
    WML_ADLSequenceValue ::= "<" ">"
                    | "<" PropertyValueDeclaration ( "," PropertyValueDeclaration )*
                      ">"
      WML_ADLRecordValue ::= ( "[" RecordFieldValue ( ";" RecordFieldValue | ";" )* "]" | "[" "]" )
          RecordFieldValue ::= <IDENTIFIER> ( ":" PropertyTypeDescription )? "="
                      PropertyValueDeclaration
  RepresentationDeclaration ::= <REPRESENTATION> ( <IDENTIFIER> "=" )? "{"
                      SystemDeclaration ( BindingsMapDeclaration )? "}" ( ";" )?
    BindingsMapDeclaration ::= <BINDINGS> "{" ( BindingDeclaration )* "}" ( ";" )?
        BindingDeclaration ::= ( <IDENTIFIER> "." )? <IDENTIFIER> "to"
                      ( <IDENTIFIER> "." )? <IDENTIFIER> ( "{"
                      ( PropertyDeclaration | PropertiesBlock )* "}" )? ";"
  DesignAnalysisDeclaration ::= ( ( <EXTERNAL> ( <DESIGN> )? <ANALYSIS>
                      <IDENTIFIER> "(" FormalParams ")" ":"
                      ( PropertyTypeDescription | <COMPONENT> | <GROUP> |
                      <CONNECTOR> | <PORT> | <ROLE> | <SYSTEM> |
                      <ELEMENT> | <TYPE> ) "=" JavaMethodCallExpr ";" ) |
                      ( ( <DESIGN> )? <ANALYSIS> <IDENTIFIER> "("
                      FormalParams ")" ":" ( PropertyTypeDescription |
                      <COMPONENT> | <GROUP> | <CONNECTOR> |
                      <PORT> | <ROLE> | <SYSTEM> | <ELEMENT> |
                      <TYPE> ) "=" DesignRuleExpression ";" ) )
       parse_DesignRule ::= ( <DESIGN> )? ( <INVARIANT> | <HEURISTIC> )
                      DesignRuleExpression ( <PROPBEGIN>
                      parse_PropertyDescription ( ";" parse_PropertyDescription | ";" )*
                      <PROPEND> )? ";"
    DesignRuleExpression ::= QuantifiedExpression
                    | BooleanExpression
    QuantifiedExpression ::= ( ( <FORALL> | <EXISTS> ( <UNIQUE> )? )
                      <IDENTIFIER> ( ( ":" | <SET_DECLARE> ) ( Type |
                      lookup_arbitraryTypeByName ) )? <IN> ( SetExpression |
                      Reference ) "|" DesignRuleExpression )
      BooleanExpression ::= OrExpression ( <AND> OrExpression )*
           OrExpression ::= ImpliesExpression ( <OR> ImpliesExpression )*
      ImpliesExpression ::= IffExpression ( <IMPLIES> IffExpression )*
           IffExpression ::= EqualityExpression ( <IFF> EqualityExpression )*
      EqualityExpression ::= RelationalExpression ( <EQ> RelationalExpression | <NE>
                      RelationalExpression )*
    RelationalExpression ::= AdditiveExpression ( "<" AdditiveExpression | ">"
```

```
                    AdditiveExpression | <LE> AdditiveExpression | <GE>
                    AdditiveExpression )*
AdditiveExpression ::= MultiplicativeExpression ( <PLUS> MultiplicativeExpression |
                    <MINUS> MultiplicativeExpression )*
MultiplicativeExpression ::= UnaryExpression ( <STAR> UnaryExpression | <SLASH>
                    UnaryExpression | <REM> UnaryExpression )*
UnaryExpression ::= <BANG> UnaryExpression
                  | <MINUS> UnaryExpression
                  | PrimitiveExpression
PrimitiveExpression ::= "(" DesignRuleExpression ")"
                  | LiteralConstant
                  | Reference
                  | SetExpression
Reference ::= <IDENTIFIER> ( ( "." <IDENTIFIER> ) | ( "." <TYPE> ) |
                    ( "." <COMPONENTS> ) | ( "." <CONNECTORS> ) | ( "."
                    <PORTS> ) | ( "." <ROLES> ) | ( "." <MEMBERS> ) | ( "."
                    <PROPERTIES> ) | ( "." <REPRESENTATIONS> ) | ( "."
                    <ATTACHEDPORTS> ) | ( "." <ATTACHEDROLES> ) )*
                    ( "(" ActualParams ")" )?
JavaMethodCallExpr ::= <IDENTIFIER> ( "." <IDENTIFIER> )* "(" ActualParams ")"
LiteralConstant ::= ( <INTEGER_LITERAL> )
                  | ( <FLOATING_POINT_LITERAL> )
                  | ( <STRING_LITERAL> )
                  | ( <TRUE> )
                  | ( <FALSE> )
                  | ( <COMPONENT> )
                  | ( <GROUP> )
                  | ( <CONNECTOR> )
                  | ( <PORT> )
                  | ( <ROLE> )
                  | ( <SYSTEM> )
                  | ( <ELEMENT> )
                  | ( <PROPERTY> )
                  | ( <INT> )
                  | ( <FLOAT> )
                  | ( <STRING> )
                  | ( <BOOLEAN> )
                  | ( <ENUM> )
                  | ( <SET> )
                  | ( <SEQUENCE> )
                  | ( <RECORD> )
ActualParams ::= ( ActualParam ( "," ActualParam )* )?
FormalParams ::= ( FormalParam ( "," FormalParam )* )?
ActualParam ::= DesignRuleExpression
```

```
FormalParam ::= <IDENTIFIER> ("," <IDENTIFIER> )* ":" ( <ELEMENT>
                | <SYSTEM> | <COMPONENT> | <CONNECTOR> |
                <PORT> | <ROLE> | <TYPE> | <PROPERTY> |
                <REPRESENTATION> | <ANY> |
                NonPropertySetTypeExpression | PropertyTypeDescription )
NonPropertySetTypeExpression ::= <SET> "{" ( <ELEMENT> | <SYSTEM> |
                <COMPONENT> | <CONNECTOR> | <PORT> |
                <ROLE> | <TYPE> | <PROPERTY> |
                <REPRESENTATION> | <ANY> ) "}"
SetExpression ::= ( LiteralSet | SetConstructor )
LiteralSet ::= ( "{" "}" | "{" ( LiteralConstant | Reference ) ( ","
                ( LiteralConstant | Reference ) )* "}" )
SetConstructor ::= ( "{" <SELECT> <IDENTIFIER> ( ":"
                lookup_arbitraryTypeByName )? <IN> ( SetExpression |
                Reference ) "|" DesignRuleExpression "}" | ( "{" <COLLECT>
                <IDENTIFIER> "." <IDENTIFIER> ":"
                lookup_arbitraryTypeByName "." lookup_arbitraryTypeByName
                <IN> ( SetExpression | Reference ) "|" DesignRuleExpression
                "}" ) )
RecordType ::= <RECORD> "[" RecordItem ( "," RecordItem )* "]"
RecordItem ::= <IDENTIFIER> ":" Type
SetType ::= <SET> "{" Type "}"
SequenceType ::= <SEQUENCE> "{" Type "}"
Signature ::= Type "<->" Type
Type ::= ( <IDENTIFIER> ( "." <IDENTIFIER> )* )
PrimitiveType ::= <COMPONENT>
                | <GROUP>
                | <CONNECTOR>
                | <PORT>
                | <ROLE>
                | <SYSTEM>
Element ::= ( <IDENTIFIER> ( "." <IDENTIFIER> )* )
                | CompoundElement
CompoundElement ::= Set
                | Record
                | Sequence
Set ::= "{" Element ( "," Element )* "}"
Record ::= "[" <IDENTIFIER> "=" Element ( "," <IDENTIFIER> "="
                Element )* "]"
Sequence ::= "<" Element ( "," Element )* ">"
```

A.6 Syntax of AADL

```
AADL ::= AADLModel <EOF>
AADLModel ::= AgentTemplateList AgentDef AgentLinkDef
```

AgentTemplateList ::= <AgentTypeList> "CML_Spec" AgentTypeDef
AgentTypeDef ::= (<AgentType> AgentTypeName <is> State Port Service)*
AgentTypeName ::= <IDENTIFIER>
State ::= "CML_Spec"
Port ::= "CML_Spec"
Service ::= <Requires> (ServiceList)? ";" <Provides> (ServiceList)? ";" <Tasks> (TaskList)? ";"
ServiceList ::= "CML_Spec"
TaskList ::= "CML_Spec"
AgentDef ::= <Agent_instances> <are> ":" (AgentName <Instantiates> AgentLinkDef ";")*
AgentName ::= <IDENTIFIER>
AgentLinkDef ::= Link ("," Link)*
Link ::= One_OneLink
| One_ManyLink
| Many_OneLink
One_OneLink ::= ProvideService "->" RequestService
One_ManyLink ::= ProvideService "->" RequestServiceList
Many_OneLink ::= ProvideServiceList "->" RequestService
ProvideServiceList ::= (ProvideService)*
RequestServiceList ::= (RequestService)*
ProvideService ::= <IDENTIFIER>
RequestService ::= <IDENTIFIER>

A.7 Syntax of FDL in WML

FDL ::= FeatureModel <EOF>
FeatureModel ::= (FeatureDef)? (Constraint)?
FeatureDef ::= FeatureName ":" FeatureExp
FeatureName ::= CompositeFeatureName
| AtomicFeatureName
CompositeFeatureName ::= <S_Composite_IDENTIFIER>
AtomicFeatureName ::= <S_Atomic_IDENTIFIER>
FeatureExp ::= <All> FeatureList
| <Oneof> FeatureList
| <Moreof> FeatureList
| FeatureName
| FeatureExp "?"
Constraint ::= DiagramConstraint
| UserConstraint
DiagramConstraint ::= AtomicFeatureName <Requires> AtomicFeatureName
| AtomicFeatureName <Excludes> AtomicFeatureName
UserConstraint ::= <Include> AtomicFeatureName

| <Exclude> AtomicFeatureName

Appendix B: *Meta*WML Libraries

This section contains the overview of *Meta*WML libraries, including commands and utilities.

B.1 Navigation Command on AST

Command: @New_Model(<list>)

Parameters:

list = The new current model

Description: Sets the current Model to the provided list.

Example(s): @New Model (M)

Command: @Model

Description: Returns the current model.

Command: @I

Description: Returns the current model item.

Command: @Posn

Description: Returns the current position in the abstract syntax tree.

Command: @Parent

Description: Returns the parent of the current model item.

Command: @GParent

Description: Returns the grandparent of the current model item.

Command: @Make(<type>, <value>, <comps>)

Parameters:

type = The specific type to add

value = The value (if component is a leaf of the AST)

comps = The subcomponents (if component is not a leaf of the AST)

Description:

Adds a AST item to the current position in the tree.

Command: @Make_Name(<string>)

Parameters:

string = The caption of the item

Description:

Converts a string to a "name" suitable to use as the value for various item types.

Command: FILL <general_type> <item_schema> ENDFILL

Parameters:

general_type = The general type to start with

item_schema = The model to parse

Description:

To return an abstract syntax tree of the given model (specified by the first parameter), starting with the given general type as specified by the second parameter.

Command: @Parse_File "<wmlfile>.wml" <specific_type>

Parameters:

<wmlfile.wml> = A WML file located in the file system

<specific_type> = The specific type to start with

Description:

To return an abstract sysntax tree of the given model (specified by the first parameter), starting with the given specific type as specified by the second parameter.

Command: @UP, @DOWN, @LEFT, @RIGHT, @GOTO

Description:

To travel through the tree if the desired position is known.

Command: @UP ?, @DOWN ?, @LEFT ?, @RIGHT ?, @GOTO ?

Description:

To check if a step is possible the transformation program can use the commands.

Command: @Edit, @Undo_Edit, @End_Edit

Description:

@Edit can alter the current item as desired. If an error occurs, @Undo_Edit can restore the original Model. An @End_Edit will commit the changes.

Command: @Splice_Over, @Splice_Before, @Splice_After

Description:

New items can be inserted via the @Splice_Over(<Items>) overwriting the current item or with @Splice_Before(<Items>) and @Splice_After(<Items>) to the left or right of the current item.

Command: @Delete, @Clever_Delete

Description:

Items can be deleted with the @Delete command. @Clever_Delete command is provided that can delete the current item and "fix up" the syntax of the resulting model.

Command: @Cut, @Buffer, @Paste_Over(@Buffer), @Paste_Before(@Buffer), @Paste_After(@Buffer)

Description:

Transformation engine can also do copy & paste operations with @Buffer command.

@Cut fills with buffer which deletes the current item and stores it in the buffer.

@Paste_Over(@Buffer) overwrites the current item.

@Paste_Before(@Buffer) adds buffer as new sibling to the left of the current item

@Paste_After(@Buffer) adds buffer as new sibling to the right of the current item.

B.2 Query Facilities

Query Functions	Return Types	Descriptions
@Class_Query (Class c)	True/False	Judge whether class c is indeed a class
@Class_Query (Class c, Method m)	True/False	Judge whether method m belongs to class c
@Class_Query (Class c, Attribute a)	True/False	Judge whether attribute a belongs to class c
@Class_Query()	Name List	Return all the class names in the system into a name list
@SuperClass_Query (Class c)	NameList	Return father names of a class (Return NULL if there is no father)
@SubClass_Query (Class c)	NameList	Return subclass names of a class (Return NULL if there is no child)
@Class_Substitutable (Class c1, c2)	True/False	Judge whether class c1 can be substituted by class c2
@Method_Query (Class c)	Name List	Return all the public method signatures in a class into a name list

@IsAbstract (Class c)	True/False	Judge whether a class is declared to be abstract.
@IsAbstract (Method m)	True/False	Judge whether a method is declared to be abstract.
@Attribute_Query (Class c)	Name List	Return all the public attribute names in a class into a name list
@InAssociation_Query (Class c)	Name List	Return role names of a class into a name list
@OutAssociation_Query (Class c)	Name List	Return role names of a class into a name list
@ExsitInAssociation (Class c)	True/False	Judge whether a role exits
@ExsitOutAssociation (Class c)	True/False	Judge whether a role exits
@Relationship (Class c1,c2)	Relationship List	Return all the relationship between two classes.
@ClassInstances_Retrieve (Class c)	Handle List	Return all the object handles of a class, including objects of subclass.
@Link_Retrieve (Object o1, o2)	True/False	Judge whether there is a link between o1 and o2
...	...	...
@ CompositePattern (Class c1, c2)	True/False	Judge whether CompositePattern relationship is hold for class c1 and c2
@ CompositePattern (Class c1)	Name List	Return all the composite classes into a name list
@ CompositePattern (Class c2)	Name List	Return all the component classes into a name list
@ CompositePattern	Name List	Return all the classes in CompositePattern into a name list
...	...	...

B.3 Action Primitives

Action Primitives	**Return Types**	**Descriptions**
@Add_Class (Class c)	N/A	Add a class c
@Remove_Class (Class c)	N/A	Remove a class c
@Extract_Class (Class c1, c2)	N/A	move code to a new class
@Add_Method (Class c, Method m)	N/A	Add a method to class c
@Remove_Method (Class c, Method m)	N/A	Remove a method from class c
@Move_Method (Class c1, c2, Method m)	N/A	Move a method from one class to another
@Extract_Method (Class c1, c2, Method m)	N/A	move code to a new method
@PullUp_Method (Class c, Method m)	N/A	Move a method from subclass to a superclass
...	...	...

@Create_Instance (Class c)	Object Handle	Create an object from class c
@Destroy_Instance (Object obj)	N/A	Delete an object.
@Create_Link (Object obj1, obj2)	N/A	Create a link that complies with an association.
@Destroy_Link (Object obj1, obj2)	N/A	Delete a link between tow objects.
@Destroy_Link (AssociationRole r)	N/A	Delete a link based on an association role.

B.4 Metric Functions

@Stat_Types(I): Return set of statement types appearing in I

@Total_Size(I): Total number of nodes (items) in I

@Stat_Count(I): Total number of statement items

@Gen_Type_Count(type, I): Number of occurrences of given generic type

@Spec_Type_Count(type, I): Ditto for a specific type

@McCabe(I): McCabe cyclometric complexity measure for I

@CFDF_Metric(I): Control-flow / data-flow metric for I

@BL_Metric(I): Branch-loop metric for I

@Struct_Metric(I): A weighted sum over all the items in I

...

@ WMC (I): Return weighted methods per class in I

@ DIT (I): Return depth of inheritance tree in I

@ NOC (I): Return number of children in I

@ NVC (I): Return number of variables per class in I

@ APM (I): Return average parameters per method in I

@ NOO (I): Return number of objects in I

...

@TNC(M): Return total number of classes in model M

...

@Cr (): Calculate code conciseness rate

@Er (): Calculate system efficiency rate

Appendix C: List of Publications

[1] F. Chen, S. Li and H. Yang, "An Agent-based Service Oriented Approach to Role-Based Access Controls", *2007 IEEE International Conference on Networking, Sensing and Control (ICNSC'07)*, London, UK, Apr. 2007.

[2] F. Chen, H. Yang and B. Qiao, "A Formal Model Driven Approach to Dependable Software Evolution", *30th IEEE International Computer Software and Application Conference (COMPSAC'06)*, Chicago, USA, Sep. 2006.

[3] F. Chen, S. Li and H. Yang, "Feature Analysis for Service-Oriented Re-engineering", *12th IEEE Asia-Pacific Software Engineering Conference (APSEC'05)*, Taibei, Taiwan, Dec. 2005.

[4] H. Guo, C. Guo, F. Chen and H. Yang, "Wrapping Client-Server Application to Web Services for Internet Computing", *6th IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05)*, Dalian, China, Dec. 2005.

[5] F. Chen, H. Yang, H. Guo and T. Liu, "Aspect-Oriented Programming based Software Evolution with Microsoft .NET", *21st IEEE International Conference on Software Maintenance (4 pages poster paper)*, Budapest, Hungary, Sep. 2005.

[6] S. Li, F. Chen and H. Yang, "Using Feature Oriented Analysis to Recover Legacy Software Design for Software Evolution", *7th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, Taibei, Taiwan, Jul. 2005.

[7] R. Liu, F. Chen and H. Yang, "Agent-based Web Services Evolution for Pervasive Computing", *11th IEEE Asia-Pacific Software Engineering Conference (APSEC'04)*, Busan, Korea, Dec. 2004.

[8] F. Chen, H. Guo and H. Yang, "Agentification for Web Service", *28th IEEE*

International Computer Software and Application Conference (COMPSAC'04), Hong Kong, Sep. 2004.

[9] F. Chen, H. Guo, L. Dai and H. Yang, "An Application Framework for Ontology-based Data Mining", *Journal of Dalian University of Technology,* Suppl., 43(S1), Oct. 2003, pp. 143-145.

[10] Y. Wang, F. Chen, L. Xu, H. Guo, J. Wan, "An Implementation of a Multi-Interface Virtual Real-Time Operating System on Windows Platform", *Journal of Dalian University of Technology,* Suppl., 43(S1), Oct. 2003, pp. 100-102 (in Chinese).